Analytics of Condition-Effect Rules

Saurabh Fadnis

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 17.02.2019

Supervisor

Dr. Tomi Janhunen

Advisor

Dr. Jussi Rintanen

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Saurabh Fadnis

Title Analytics of Condition-Effect Rules

Degree programme Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems **Code of major** ELEC3025

Supervisor Dr. Tomi Janhunen

Advisor Dr. Jussi Rintanen

Date 17.02.2019 **Number of pages** 63 **Language** English

Abstract

This thesis studies properties such as confluence and termination for a rule model with condition-effect rules. A rule model is first defined and the complexity of solving these problems is analysed. Analysis of both confluence and termination shows that they are PSPACE-complete for our rule model. We give algorithms for testing these properties. We also study certain syntactic and structural restrictions under which these problems become easier and can be solved in polynomial time for practical purposes.

Keywords condition, effect, rules, confluence, termination

Preface

I want to thank Dr. Tomi Janhunen and Dr. Jussi Rintanen for their excellent guidance. I also want to thank my colleagues Lukas Ahrenberg, Mika Parikka and Masoumeh Parsa for their support.

Otaniemi, 17.02.2019

Saurabh Fadnis

Contents

Notations, Symbols and Abbreviations

Abbreviations

AI	Artificial Intelligence
CAV	Computer-Aided Verification
SAT	Satisfiability
LTL	Linear Temporal Logic
CTL	Computation Tree Logic
ECA	Event-Condition-Action
DTM	Deterministic Turing Machine
NTM	Non-Deterministic Turing Machine

Notations and Symbols

This table contains the notations used in the thesis for quick reference.

$\mathcal{A}$	Set of state variables
(p, e)	Rule
$\mathcal{R}$	Set of all rules (or forced rules) in the rule model
s	State
$\mathcal{S}$	Set of all possible states of the system
I	Initial state
$(\mathcal{A}, \mathcal{R}, I)$	Transition system
$\mathcal{R}_s$	Set of applicable rules in state s
$succ_e(s)$	Successor state on application of rule (p, e) to state s
$\mathcal{P}(s)$	Set of all possible successor states to state s
$\mathcal{U}$	Set of event rules
$(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$	Transition system with events

1 Introduction

Transition system models are widely used in describing dynamic systems and representing various kinds of software as well as hardware systems [2]. Transition systems can model telecommunication protocols, communication networks, electricity networks, embedded software programs, distributed control systems, algorithms etc. Many real world applications can be modeled by sequences of states, each obtained through systematic transitions. The state-transition models can then be analysed, verified, validated and monitored. Transition systems can also be subjected to very powerful formal analysis methods [15] which help us study their properties much more closely. Using these methods, we can also control and implement an actual process. As an example, before a piece of program or software is written, its high level transition system model can be created and analysed first [43]. If the analysis yields desired result, we can go ahead and implement the actual software. This application, describing software as a transition system, is the main motivation for this thesis.

A transition system model consists of a set of states and a set of transitions. The transitions govern how the system can move from one state to another. There are various ways in which the transitions can be defined. We analyse transition systems where transitions are condition-effect rules. These rules together form a rule model which is the main object of study in this thesis.

Rule Model

The rule model is defined by of a set of rules. Each rule in the rule model consists of two parts: the precondition and the effect.

1. The *precondition* is a Boolean combination of conditions over state variables. If the precondition is true, then the rule is applicable, i.e., it can be fired.

2. The *effect* describes what happens when the rule is fired, i.e., how state variables change as a result. In the simplest case, the effect simply consists of assignments $x := e$, where x is a state variable and e is an expression of the same type as the state variable.

In this rule model, the system has an initial state and other states that are obtained from this initial state by sequences of rule firings (transitions). Some rule, whose precondition is satisfied in the initial state, is fired first. The effect of this rule can further make other rules applicable, which can then be fired, leading to potentially long sequence of rule firings. Section 3 presents the terminology and the formal definitions required to define the rule model. We study the case in which if multiple rules are applicable, one of the rules is fired non-deterministically. This firing rule may dynamically cause some other rules to be applicable. In our case, one of the applicable rules

must always fire until none of the rules remain applicable. In other words, the system should always have a maximal execution. As this is non-deterministic, there can be multiple sequences of possible executions. Different orders of rule executions may lead the system into different terminal states, if the executions terminate at all. The non-deterministic behaviour makes the system quite unpredictable. Our interest lies in predicting this behaviour, in terms of properties such as confluence and termination.

Confluence is the property of a system that all possible sequences of rule firings lead to the same state. Without this property, the choice of the next rule to be fired is critical in the sense that it may affect which terminal state is reached. An example of this can be seen when a university student has to study certain required courses in order to graduate. The order in which the required courses are done does not make a difference in the end result, i.e., graduation. At any point, the student has a plethora of courses he can do, but can select only a few every semester. Sometimes, there is some internal order required among few courses but ultimately whatever way he selects, he will still graduate after he completes the required credits, making this system confluent. If this system was not confluent, the student might never be able to graduate if he selects one wrong course. So, confluence is quite often a desired property. This makes studying confluence property for complex transition systems, before implementation in real world processes, both necessary and quite interesting. The confluence property is often expressed as follows: If states s_1 and s_2 are reachable from the initial state, then there is a state s_3 that is reachable from both s_1 and s_2. Confluence properties have earlier been investigated mostly in connection with algebraic and equational reasoning, and rewriting systems [20].

Another important property we analyse is *termination*. Terminating systems have the property that all possible sequences of rule firings are finite. This means all maximal executions in a terminating system are finite. Just like confluence, this is usually a desired property. If we model the university student from the above example and write a rule that the student can pick a course again if it is available, there is a possibility that the student keeps picking the same course every year and never graduates. This analogy can be applied to complex algorithms and programs as well. If they keep doing certain action again and again, and keep returning to the same state again and again, they may never terminate. For a system defined by the rule model to be terminating, it should terminate in every possible rule execution sequence. Another important property to analyse, about sequences of rule firings, is their length. We are interested in finding a finite upper bound on the length of rule firings. If there is some finite bound N on the lengths of rule firing sequences, it may also be important to find as tight such N as possible.

For these properties, we are interested in the following:

- **Computational complexity:** What are the computation resources (memory, time) that are needed in determining whether the property in question holds.
- **Algorithms:** We want to develop practically efficient algorithms for testing

these properties, even when testing them is theoretically, in the worst case, very difficult.

- **Special cases:** We want to identify syntactical and structural features that make it easier to test these properties. We would like to have these tests run in polynomial time, which is not in general achievable.

We look into the confluence and termination properties for systems with an evolving state in Section 4. We prove that both confluence and termination properties are PSPACE-complete for our rule model.

After studying these properties with a theory-oriented view, we move towards more practical aspects of these in Section 5. In the framework that motivates our work, the rule firings to be analysed are triggered by specific events, and confluence and termination problems need to be analysed only for states which immediately follow these events. We incorporate this into our rule model, so now, in the rule model, none of the rules are applicable initially. Then some event occurs (e.g., user action) which makes some rules applicable. One of the applicable rules is always fired until no rules are further applicable. This repeats the next time some event occurs. We prove that this addition to the rule model does not affect the complexity of confluence and termination problems.

Finally, in Section 6 we look at some special cases in which these intractable problems of confluence and termination can be solved in PTIME. Under some special and restricted cases, the algorithms and results from this section can be applied in real world systems to make confluence and termination problems tractable.

2 Background

In this section, we first look into some of the previous research regarding transition systems and what kind of problems have been analyzed using these transition system models. We also explain our main motivation behind this thesis, and then look at the research areas where confluence and termination problems have been studied previously.

2.1 Transition System Models

In the field of computer science, transition system models have a very prominent place in AI (Artificial Intelligence) and CAV (Computer-Aided Verification) research. Planning and decision-making in AI deals with deciding what sequences of actions to take, so that the system reaches the goal or goal state. STRIPS planner [25], which became quite successful in 1970's, used state-transition paradigm where transitions were defined through STRIPS operators. In planning problem, a plan can only exist if the goal state is reachable, i.e., there exists a sequence of transitions which can take the system from an initial state to the goal state. Testing reachability is an integral part of any planning problem. A useful result regarding reachability is that it has been proven to be PSPACE-complete in general, and NP-hard or NP-complete with certain restrictions for the STRIPS planner [8]. STRIPS also paved way for other planners [7, 31] using similar transition system models. Later, the approach of using SAT (satisfiability) for state-space search in a transition model gained prominence in bounded model checking for LTL (Linear Temporal Logic) [6] and CTL (Computation Tree Logic) [12]. Model checking is used as a verification technique for verifying properties of transition system models using temporal logic to search the associated state-transition graphs [13, 14]. In computer-aided verification, reachability analysis is also quite important, especially for problems like deadlock detection and prevention (e.g., [18], [24]), and safety analysis [33]. A great tool for modeling transition systems and their verification are Petri nets [37]. Petri nets are a graphical and mathematical modeling tool for the representation of distributed systems, especially asynchronous, non-deterministic and parallel systems [34].

All this research relies on transition system models in one form or another. The applications of transition systems described here are just a small snippet of their power and flexibility. Transition systems have a much wider range of applications and have generated a lot of interest in many research areas. Unfortunately listing all those down is beyond the scope of this thesis.

In a transition system model, states can be represented as atomic objects. But

for representing very large systems, it can quickly get unwieldy and unnecessarily complex. Also, the state spaces of many systems are highly regular, so instead of listing them enumeratively, we can represent the states compactly with just the valuation of state variables. If we restrict the system to two-valued (Boolean) state variables, we need only n such state variables to represent 2^n states. The transitions between the states can then be represented in terms of changes to the values of these state variables. In our case, the transitions are condition-effect rules that describe what values the state variables must have before a rule can be applied, and how the values of the state variables change after a rule has been fired. We represent these rules succinctly as tuples (p, e) where p is the precondition and e is the effect. The precondition p is a propositional formula over state variables and e is a set of literals over state variables. This one rule can act as a transition from all states where precondition p holds to all the new states obtained by making true the literals in the effect. So a small number of rules can represent a very high number of different transitions between the states. This type of representation using valuation of state variables is called *succinct transition system* [39] and has been used in some applications, for example, in dynamic systems diagnosis [40].

2.2 Motivation

This research is part of the EIAI project at Aalto University, Finland. The EIAI project deals with automation in software production and covers all major parts of software development including database management, internal control logic, and user interfaces, for knowledge-intensive information systems. The idea behind the project is that all parts of software system are represented as condition-effect rules. These condition-effect rules make various functionalities available to the users, which the user can choose whether to apply or not. In addition to this, there are condition-effect rules that will fire without being requested by a user. These rules respond to certain changes in the software data and describe obligatory behaviours of the software system. E.g., a banking software can be set up so that bills are automatically paid whenever salary is deposited in the account every month. In this case the system does not need the user action every month for every bill payment. Instead it automatically carries out the required action in response to the change in the data that occurs when the salary is deposited.

This thesis is a preliminary investigation on the rule analytics pertaining to these condition-effect rules for the EIAI project. In this thesis, our focus is on Boolean facts, but the EIAI project itself supports a richer set of datatypes and is capable of achieving a very high degree of automation in the production of complex large-scale software systems.

2.3 Related Work

Confluence and termination have been an integral part of computational analysis and have been studied extensively for *rewriting systems* [20], but the research does not translate well to a general rule model as discussed in this thesis. Rewriting systems can be said to have evolved from lambda calculus and have been widely used in decision theory, equational reasoning, programming, automatic theorem proving, etc. [21]. In rewriting systems there are rules which replace a term on the left side with a term on the right. Here the replacements are non-deterministically chosen as more than one replacement can be possible at a time. Confluence in rewriting systems is known as Church-Rosser property. The system is said to have Church-Rosser property if the order in which the reductions are applied does not make any difference to the eventual result [11]. Confluence and Termination in rewriting systems is in general undecidable [27, 38], but it has been proven to be decidable under certain restrictions for some rewriting systems [35, 17, 19]. Dershowitz [20] gives a fairly comprehensive survey about rewriting systems, discussing both termination and confluence. In [30], results for both confluence and termination are applied from non-conditional rewriting to conditional rewriting systems. Our rule model is closest to this conditional rewriting systems in which rewriting rules are applied only if certain conditions are true. But our rule model is still vastly different and the results for rewriting systems cannot be directly applied to this.

The other areas where confluence and termination have been investigated are database management systems and Petri nets [37]. Petri nets are place/transition nets in which tokens in different places represent the state of the system. The transitions in a Petri net may fire if there are sufficient number of tokens in the transition's input place. This representation of transitions is quite close to our framework of rules and can be mapped to our condition-effect rules, but not in general vice versa, as the rules in our rule model can be extended much more easily to include complex datatypes with complex preconditions for the transitions to fire. Termination problem has been studied in context of Petri nets mostly as *fair nontermination problem* [9]. Petri nets can have some fairness conditions that govern how often the transitions are fired with respect to how often they are firable. The fair non-termination problem is the problem of determining if there exists an infinite sequence of transitions following the given fairness property. Howell et al. [26] discuss the decidability of this non-termination problem under different fairness definitions. Zimmer et al. [46] discuss termination based on the analysis of the reachability graph of a Petri net by modeling rule semantics of database system to Petri nets, but they do not go into the details about the complexity of their methods.

Confluence for Petri nets is related to *home marking problem* and it has been first studied as *directedness* in [4]. In a dynamic system, *home state* is a state which is reachable from every other reachable state. The home marking problem is the problem of testing if the given state is a home state. Studying this problem is

especially useful for cyclic processes such as operating systems, which have to reach a home state after all operation sequences. This home marking problem has been proven to be decidable [23]. Instead of finding out if the given state is a home state, we are more interested in determining if the system has any home state at all. This problem is much closer to our confluence problem and has been proven to be decidable [5]. Leahu and Tiplea [32] prove that the confluence property in connection with Petri nets is decidable and also explore its relation to term rewriting systems.

Confluence and termination analysis has also been an integral part of database management systems with active rules. Modern database management systems now have computing capabilities to automatically respond to events that happen either internally or externally to the database system. These database systems consist of a passive database and a set of active rules with ECA (Event-Condition-Action) paradigm quite similar to the ones in the rule model we discuss. The database automatically takes required actions after an event occurs if certain conditions are met. It is quite important in these systems that the rule execution terminates and also that the same set of rules leads to the same terminal state in all possible execution sequences. We are interested in static analysis, i.e., analysis done before the system is in execution. Aiken et al. [1] discuss these properties of confluence and termination for a state-transition rule system called Starburst using the method of triggering graphs. Priority among rules is assumed in their discussion and commutativity among rules, i.e., rules which do not affect one another, is first discussed in that paper. Voort and Siebes [44] also make similar analysis for different semantics based on how the rules trigger when an event occurs. Wang et al. [45] investigate the confluence property in active databases with meta-rules (rules which govern the interaction between other rules) and determine the complexity of testing confluence. This is related to our work as, similar to our model, it has dynamically changing system in which execution of one rule may cause other rules to be firable. But non-determinism in rule firing is the main focus in this thesis, while the inclusion of meta-rules takes this non-determinism away. The meta-rules control the interactions and relationships among different rules, and when multiple rules are firable, it determines which rule should be fired. The meta-rules also help determine, in polynomial time, whether a rule will never execute, or if only some rules can ever execute, a task much more complex without the determinism of the meta-rules, as we will see once we define the rule model. The groundwork for meta-rules and static database system has been laid in [28]. Decidability and undecidability of termination problem with respect to various languages for active database is discussed by Bailey et al. [3]. Termination (or halting problem) is a classic example of an undecidable problem, but it is decidable for a linear bounded automaton (a Turing machine with finite space) as there are only limited number of configurations to test. So decidability of termination depends on how the system and rule model are defined. Comai and Tanca [16] translate the set of rules from different rules models to an internal format consisting of logical clauses and amalgamate the analyses of confluence and termination with rule prioritization. Efforts have also been made for applying results of Petri nets to database models by converting the active rules to transitions in Petri net [10, 29].

As we see here, priority among rules, which makes the system more deterministic, has been a recurring theme among the research about active rules. In this thesis our focus in on non-determinism. Also, the confluence and termination properties studied previously have not quite been in the same context as in this thesis. Our goal is in describing software as a transition system, where the rules describe steps of the computation, and analysing this transition system so as to provide better insight to the developer in regards to the behaviour of the software.

3 The Rule Model

This section presents the Boolean rule model that has been used for all the analysis done in the thesis. Other terminology and the formal definitions used throughout the thesis are also defined in this section. As mentioned in the introduction, a rule consists of a precondition and an effect. To define Boolean rule model, we only consider Boolean variables, but this rule model can be easily extended to encompass other datatypes like integers, reals etc. as well. Some definitions presented in this section are directly inspired from succinct transitions system [39].

3.1 Propositional Formulas

Let X be a set of propositional variables (atomic propositions). These variables must take value of either TRUE or FALSE. Propositional formulas are constructed from atomic propositions by using logical connectives. The symbols $\wedge$, $\vee$ and $\neg$ are logical connectives denoting the conjunction, disjunction and negation respectively.

Definition 1 (Propositional formulas). If X is a set of propositional variables, the set of propositional formulas is defined inductively as follows:

1. For all $x \in X$, x is a propositional formula.
2. The symbols $\bot$ and $\top$, respectively denoting truth-values FALSE and TRUE, are propositional formulas.
3. If ϕ is a propositional formula, then so is $\neg\phi$.
4. If ϕ and ϕ' are propositional formulas, then so is $\phi \vee \phi'$.
5. If ϕ and ϕ' are propositional formulas, then so is $\phi \wedge \phi'$.

We define the implication $\phi \rightarrow \phi'$ as an abbreviation for $\neg\phi \vee \phi'$, and the equivalence $\phi \leftrightarrow \phi'$ as an abbreviation for $(\phi \rightarrow \phi') \wedge (\phi' \rightarrow \phi)$.

Definition 2 (Literal). A *literal* is either a propositional variable or its negation. If X is a set of propositional variables, then for all $x \in X$, the propositional variable x and the negated propositional variable $\neg x$ are literals.

We define the complements of literals as $\overline{x} = \neg x$ and $\overline{\neg x} = x$ for all $x \in X$.

Definition 3 (Valuation of propositional variables). A *valuation* $v : X \to \{0, 1\}$ is a mapping from propositional variables $X = \{x_1, ..., x_n\}$ to truth-values 0 and 1. The values 0 and 1 correspond to FALSE and TRUE, respectively. For a propositional variable $x \in X$, we define $v \models x$ if and only if $v(x) = 1$.

If $\mathcal{L}$ is a set of propositional formulas over X, valuations can be extended to formulas $\phi \in \mathcal{L}$, i.e., the valuation function is given by $v : \mathcal{L} \to \{0, 1\}$.

Definition 4 (Valuation of propositional formulas). A given valuation $v : X \to \{0, 1\}$ of atomic propositions can be extended to a valuation of arbitrary propositional formulas ϕ and ϕ' over X as follows:

1. $v(\neg\phi) = 1$ iff $v(\phi) = 0$
2. $v(\top) = 1$
3. $v(\bot) = 0$
4. $v(\phi \wedge \phi') = 1$ iff $v(\phi) = 1$ and $v(\phi') = 1$
5. $v(\phi \vee \phi') = 1$ iff $v(\phi) = 1$ or $v(\phi') = 1$
6. $v(\phi \to \phi') = 1$ iff $v(\phi) = 0$ or $v(\phi') = 1$
7. $v(\phi \leftrightarrow \phi') = 1$ iff $v(\phi) = v(\phi')$

3.2 Transition System

We define our rule model as a transition system $(\mathcal{A}, \mathcal{R}, I)$ where $\mathcal{A}$ is a finite set of propositional variables to represent states of the system, $\mathcal{R}$ is a finite set of rules which act as transitions and I is the initial state.

The states of the system are represented as the valuation of propositional variables in $\mathcal{A}$. Since the variables represent states, we call these variables as state variables. They take the values TRUE or FALSE.

Definition 5 (State). Let $\mathcal{A}$ be a finite set of state variables. Each state s is defined as a valuation of $\mathcal{A}$.

$$s : \mathcal{A} \to \{0, 1\}$$

As we identify states with valuations of state variables, we can also use propositional formulas over these state variables to identify sets of states. The set of all possible states is represented by $\mathcal{S}$. The initial state of the system is denoted by I.

We can also represent actions or effects of rules in our rule model as changes to the valuation of state variables. Rules are the transitions in our rule model and firing them changes the state of the system. This brings us to the definition of rules.

Definition 6 (Rule). Let $\mathcal{A}$ be a finite set of state variables. A *rule* is defined as a tuple (p, e) where

1. p is a propositional formula over $\mathcal{A}$, called the *precondition*, and
2. e is a finite set of literals over $\mathcal{A}$, called the *effect* of the rule.

We denote the set of all the rules by $\mathcal{R}$.

When the valuation of the propositional formula p is 1 in a state s, it means that the precondition is TRUE. In that case, (p, e) can be an applicable rule in state s.

Definition 7 (Applicable rules in state s). For a particular state s, all rules and only those rules are applicable for which

1. $s \models p$ (p is true in state s),
2. for each $l \in e$, $\bar{l} \notin e$ (effects are consistent), and
3. for some $l \in e$, $s(l) = 0$ (the effect should change at least one state variable).

The 2^{nd} point states that the literals in the effect should not contradict each other, and to elaborate on 3^{rd} point a bit more, an applicable rule should be capable of changing the current state. If literals in the rule's effect are already true, it is irrelevant if it fires or not, as it will not change the state of the system. So, in our rule model, we do not consider it as an applicable rule.

The set of all applicable rules in state s is denoted by $\mathcal{R}_s$.

$$\mathcal{R}_s \subseteq \mathcal{R}$$

An applicable rule (p, e) in state s is applied or fired by making TRUE all literals in the set e of the rule and retaining the values of state variables for state s not occurring in e. A fired rule always changes the state of the system, as only those rules which can change the state of the system can be applicable (see 3^{rd} point in

the definition of applicable rules). The new state obtained is called the successor state.

When multiple rules are applicable, one of the rules is arbitrarily chosen to be fired. This is where the non-determinism comes into the picture. This is true for systems where there is no defined priority among rules.

Definition 8 (Successor of a state with respect to a rule). Let $\mathcal{A}$ be a finite set of state variables. State s' is a *successor* of state s with respect to the applicable rule $(p, e) \in \mathcal{R}_s$ where

1. $s' \models e$, and
2. $s'(a) = s(a)$ for all $a \in \mathcal{A}$ that do not occur in e.

The set of all possible successors s' from all applicable rules in the state s is denoted by $\mathcal{P}(s)$. Each state s can have at most one successor state s' for each rule in $\mathcal{R}_s$.

The rule (p, e) can be thought of as a function which takes a state as an input and applies the effect e. The state s' obtained on application of rule (p, e) to state s is denoted by function $succ_e(s)$:

$$succ_e(s) = s'.$$

In the successor state, new rules can become applicable which can then be fired to make other rules applicable resulting in a longer execution sequence. An execution is a sequence of states such that each consecutive state is obtained by firing one of the applicable rules in the previous state.

Definition 9 (Execution). An *execution* from state s_1 to s_n is an ordered sequence of states $(s_1, s_2, \ldots, s_i, \ldots, s_n)$ such that there exists a sequence of rules $((p_1, e_1), (p_2, e_2), \ldots, (p_i, e_i), \ldots, (p_{n-1}, e_{n-1}))$ for all $1 \leq i \leq n-1$, where

1. $(p_i, e_i) \in \mathcal{R}_{s_i}$, and
2. $s_{i+1} = succ_{e_i}(s_i)$.

In an execution, each state can also be said to be reachable from any of the previous states by firing the rules in the order required by the execution.

Definition 10 (Reachable state). A state s' is *reachable* from s if and only if there is an execution $(s, \ldots, s')$ from state s to state s'.

The set of all the states reachable from s is denoted by $\mathcal{C}(s)$. So if s' is reachable from s, we denote it as $s' \in \mathcal{C}(s)$.

Definition 11 (Infinite execution). An *infinite execution* is an ordered sequence of states $(s_1, s_2, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots)$ such that, there exists a corresponding sequence of rules $((p_1, e_1), (p_2, e_2), \ldots, (p_{i-1}, e_{i-1}), (p_i, e_i), (p_{i+1}, e_{i+1}), \ldots)$, where

1. $(p_i, e_i) \in \mathcal{R}_{s_i}$, and
2. $s_{i+1} = succ_{e_i}(s_i)$.

Definition 12 (Maximal execution). An execution is *maximal* if either of the following holds.

1. The execution is finite, hence of the form $(s_1, s_2, \ldots, s_n)$, and $\mathcal{P}(s_n) = \emptyset$. There are no applicable rules in the final state of the execution.
2. The execution is infinite.

Every transition system has at least one maximal execution. One of the applicable rules is always fired until there are none left. The system might have an infinite maximal execution if there is always some applicable rule to fire. This infinite execution is not desirable, so we are more interested in determining whether the system always terminates.

3.3 Properties of the System

Now we look at the properties of the system we are interested in studying.

Definition 13 (Terminating system). A system is *terminating* if there is no infinite execution sequence in the system, or in other words, all maximal executions are finite.

Non-terminating systems have at least one infinite maximal execution sequence. In terminating systems, we are also interested in finding how many rules are fired before the system terminates or if there is some finite bound N on the lengths of rule firing sequences. It may also be important to find as tight N as possible.

Definition 14 (Bound of a terminating system). A terminating system $(\mathcal{A}, \mathcal{R}, I)$ is *N-bounded*, if there is no execution $(I, s_1, ..., s_n)$ in the system, such that $n > N$.

Another important property we look into is confluence. This property is often expressed as follows: If states s_1 and s_2 are reachable from the initial state I, then there is always a state s_3 that is reachable from both s_1 and s_2. A common state should be reachable from any other reachable state in the system for it to be confluent.

Definition 15 (Confluent system). A system $(\mathcal{A}, \mathcal{R}, I)$ is *confluent* if for all $s_1, s_2 \in \mathcal{C}(I)$, there exists a state s_3 such that $s_3 \in \mathcal{C}(s_1)$ and $s_3 \in \mathcal{C}(s_2)$.

Note that the system need not be terminating for it to be confluent or confluent for it to be terminating. A non-terminating system can be confluent if there is always a unique reachable state from all the states even if the system is stuck in a loop, which is an infinite execution. This is shown in Figure 1(a). There is a loop in the system but the final state s_f is always reachable. While Figure 1(b) shows two possible final states, but without any loop formed. The system terminates but it might end up in any of the final states, so it is not confluent.

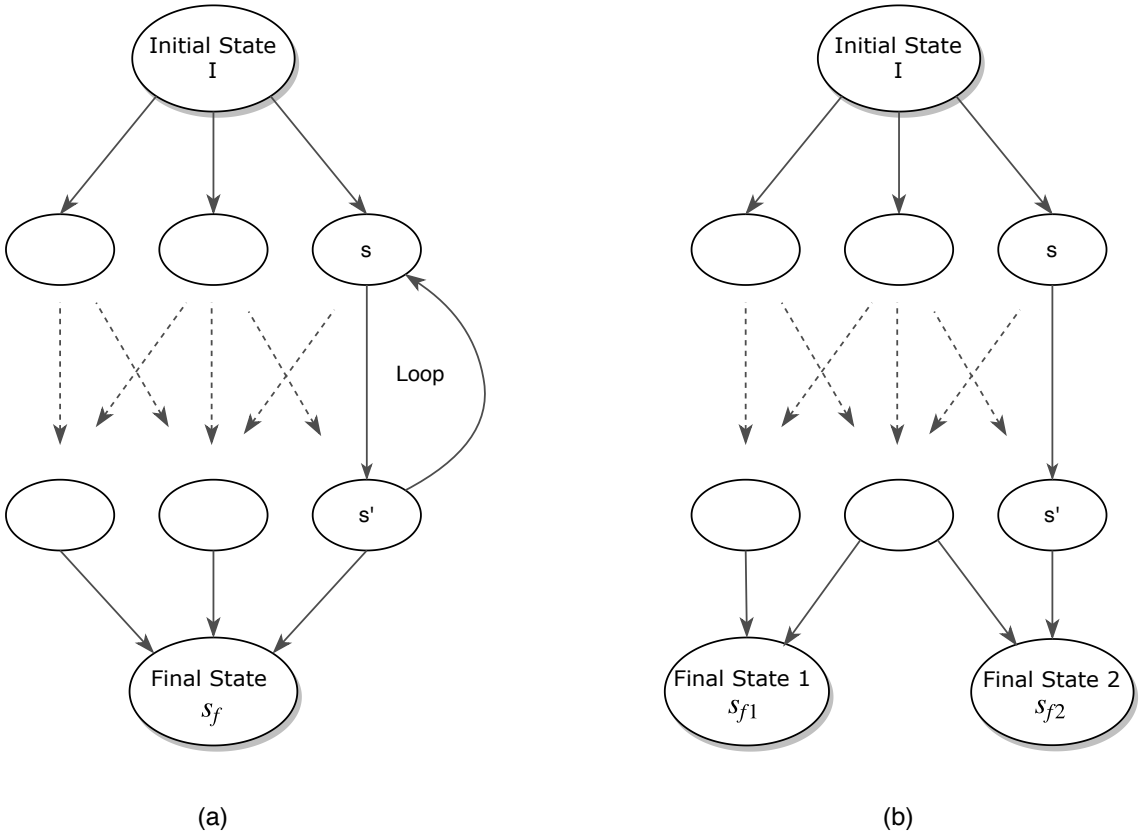

Figure 1: (a) Non-terminating, confluent system (b) Terminating, non-confluent system

Definition 16 (Convergent system). A system which is both terminating and confluent is a *convergent system.*

Our focus in this thesis is more on convergent systems as the confluence property in a terminating system is desirable. Without this property, the choice of the next rule to be applied is critical in the sense that it may affect which terminal state is

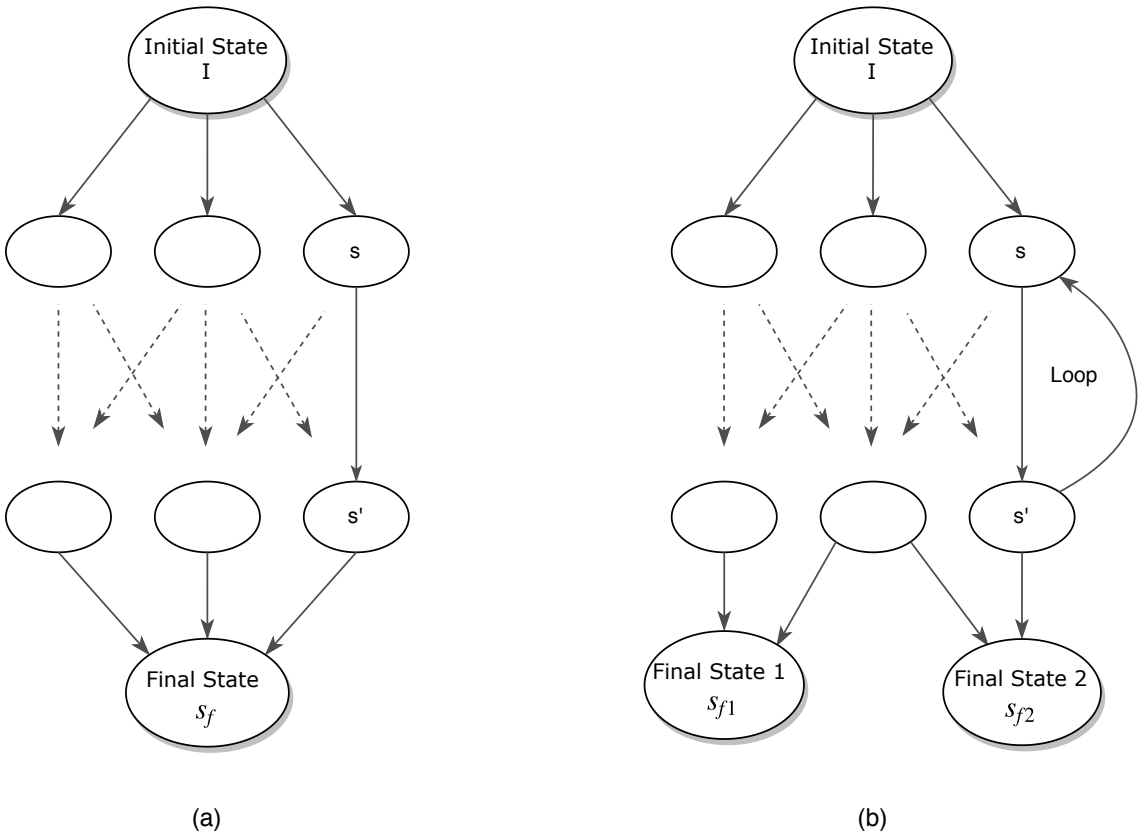

Figure 2: (a) Convergent system (b) Non-terminating, non-confluent system

reached. A terminating rule set is confluent if for an initial state, the order of rule execution does not influence the final state of the system and there is only one final state.

Figure 2(a) shows an example of convergent system. It is both confluent and terminating. Figure 2(b) is neither confluent nor terminating.

In the next section we analyse these properties for the system with some rules applicable in the initial state and then other states are obtained from this initial state by a sequence of rule firings. In our case, in every state, one of the applicable rules must fire, and the firing sequence stops only when and if the system reaches a state with no applicable rules.

4 Analysis

In this section we discuss and prove some important results about confluence and termination using the definitions and the rule model discussed in the previous section. First let us look briefly at various complexity classes.

4.1 Complexity Classes

In this subsection, we will define Turing machines, and go over preliminaries of time and space complexity classes such as P, NP, PSPACE and NPSPACE. For a detailed introduction to computational complexity, see any of the standard textbooks like C. H. Papadimitriou [36], or M. Sipser [42].

Definition 17 (Turing machine [42]). A *Turing machine* is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the blank symbol #,
3. Γ is the tape alphabet, where $\# \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\textit{move left, move right}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accepting state, and
7. $q_{reject} \in Q$ is the rejecting state, $q_{reject} \neq q_{accept}$.

In theory of computational complexity, a decision problem is a computational problem that has either a yes or a no as an answer for a given input. The decision problem is called solvable or decidable if a Turing machine halts and answers correctly on all possible instances of the problem. The input for the Turing machine is a string on the tape and the answer is yes if the Turing machine reaches an accepting state or a no if it reaches a rejecting state.

For time complexities, class P (or PTIME) consists of all decision problems solvable in polynomial computation time by a deterministic Turing machine (DTM), while class NP consists of all decision problems solvable by a non-deterministic Turing machine (NTM) in polynomial time. Language L' is said to be polynomial time many-one reducible to L, if there exists a polynomial time computable function f

where for every x, $x \in L'$ *iff* $f(x) \in L$. For complexity class C, a problem L is called *C-hard* if all problems $L' \in C$ are polynomial time many-one reducible to L. Problem L is *C-complete* if it belongs to class C and is also C-hard.

For space complexities, PSPACE is the class of decision problems solvable in polynomial space on a deterministic Turing machine, while class NPSPACE is the class of decision problems solvable in polynomial space on a non-deterministic Turing machine. One important result regarding space complexities is that PSPACE = NPSPACE by Savitch's theorem [41].

The main result in this thesis is the simulation of deterministic Turing machine in polynomial space.

4.2 Confluence

First we study the property of confluence.

Definition 18 (Instance of the confluence problem). An instance of the confluence problem is specified by a tuple $(\mathcal{A}, \mathcal{R}, I)$ where $\mathcal{A}$ is a finite set of state variables, $\mathcal{R}$ is a finite set of rules, and I is the initial state given by the valuation of the state variables in $\mathcal{A}$.

We define CONFLUENCE as the decision problem of determining if an instance $(\mathcal{A}, \mathcal{R}, I)$ of confluence problem is a confluent system.

Lemma 1. *CONFLUENCE is in PSPACE.*

Proof Idea. To prove that CONFLUENCE is in PSPACE, we can reduce this problem to reachability (or planning) problem, which has been proven to be in PSPACE [8]. We can check if a common state is reachable from every other reachable state.

Proof. First we see that the problem of testing if a state s_2 is reachable from state s_1, i.e., check if $s_2 \in \mathcal{C}(s_1)$, is in PSPACE. This is similar to the STRIPS planning problem which has been proven to be PSPACE-complete.The proof is given in [8].

In our rule model, applicable rules can be non-deterministically chosen to be fired and also the number of states is bounded by the number of state variables. So, if there are n state variables, then the length of the smallest reachability path is less than 2^n as it exhausts all possible states and after this, the states are bound to repeat. If a reachable path exits it should already be found by then. So only 2^n rule

firings are required to check reachability. In this non-deterministic model only the latest state needs to be stored for all execution paths and maximum execution length can only be 2^n. If a state is reachable from any other state, it can be found on one of the execution branches. Also the space needed to store a state is polynomial in terms of state variables. As each branch of this non-deterministic model needs polynomial space, this is in NPSPACE. But Savitch's theorem [41] proves that NPSPACE = PSPACE. So reachability is in PSPACE.

The following Algorithm 1: REACH(s_1, s_2, t), based on Savitch's proof [41], solves the problem of reachability in PSPACE for our model. It tests whether a reachability path of length $\leq t$ exists. It calls itself recursively, and each level has to store only the two states and the value of t. As states are represented by state variables, only polynomial space is needed to save the states. Furthermore, the value of t is also halved in each recursion. Here t is a power of 2 and maximum value it can take is 2^n. So the space needed by the algorithm is $\mathcal{O}(\log t)$, or in other words $\mathcal{O}(f(n))$ which is polynomial in n for 2^n maximum execution length.

Algorithm 1: REACH(s_1, s_2, t)

```
if t = 1 then
    if s1 = s2 OR s2 ∈ P(s1) then
        /* check if s2 is reachable from s1 in 0 or 1 steps as t
           is a power of 2.                                      */
        return TRUE;
    else
        return FALSE;
    end
else
    for all states s' do
        /* Use recursion                                        */
        if REACH (s1, s', t/2) AND REACH (s', s2, t/2) then
            return TRUE;
        end
    end
    return FALSE;
end
```

We use this PSPACE reachability algorithm to prove that CONFLUENCE is in PSPACE.

Algorithm 2: CONF$(\mathcal{A}, \mathcal{R}, I)$ tests confluence for a system $(\mathcal{A}, \mathcal{R}, I)$. As stated above there are 2^n states possible. The set of all possible states in the system is denoted by $\mathcal{S}$. The states in $\mathcal{S}$ can be enumerated one by one from the state variables present in $\mathcal{A}$.

Algorithm 2: CONF($\mathcal{A}, \mathcal{R}, I$)

```
for all pairs of states s1 ∈ S and s2 ∈ S do
    if REACH (I, s1, 2^n) AND REACH (I, s2, 2^n) then
        is_confluent := 0;
        for all states s3 ∈ S do
            if REACH (s1, s3, 2^n) AND REACH (s2, s3, 2^n) then
                /* Set confluent flag only if the states are
                   confluent                                    */
                is_confluent := 1;
            end
        end
        /* If confluent flag is not set then the system is not
           confluent.                                           */
        if is_confluent = 0 then
            return FALSE;
        end
    end
end
return TRUE;
```

As the representation of each state in the rule model is a polynomial function of the number of state variables, each state requires only polynomial space to store. So the iteration of states for all three s_1, s_2 and s_3 takes polynomial $f(n)$ space for n state variables. Also each call to REACH takes polynomial space, as discussed earlier and in [8], during its call and returns TRUE or FALSE upon execution. As each part of the algorithm to check confluence takes only polynomial space, the problem is in PSPACE. □

Lemma 2. *CONFLUENCE is PSPACE-hard.*

Proof Idea. To prove that CONFLUENCE is PSPACE-hard, we need to show that every language in PSPACE can be reduced to CONFLUENCE, or in other words, reduce any polynomial space bounded Turing machine to our problem of CONFLUENCE in polynomial time. We can do so by encoding transitions of the Turing machine to our rule model, and making sure that it is confluent only when the Turing machine accepts and non-confluent when it rejects. This can be achieved by introducing an additional rule which offers the system an alternative execution path to the system state corresponding to the accepting state of the Turing machine.

Proof. Let M be any polynomial space bounded deterministic Turing machine which decides a language in PSPACE. As it is space bounded, let n be the length of the tape. So position of the head can be at any place $0, 1, \ldots, n-1$. If the length of input string to the Turing machine is k, n is a polynomial function $f(k)$.

We can define state variables as

- $in(i, x)$ = symbol x is in tape position i, $x \in \Gamma$, and
- $at(i, q)$ = the Turing machine is in state q and the head is in position i.

So if the Turing machine contains input on the tape from position 1 to k, the initial configuration of the Turing machine can be encoded in our rule model as,

$I = at(0, q_0),\ in(0, \#),\ in(1, x_1),\ in(2, x_2), \ldots,\ in(k, x_k),$
$in(k+1, \#),\ in(k+2, \#), \ldots, in(n-1, \#)$ variables are TRUE
... and all other state variable are FALSE.

Now, the rule model for the CONFLUENCE problem is defined by rules. So we encode transitions of the Turing machine as rules. Suppose the Turing machine is in state q, the machine head is at position $i < n-1$, the head reads alphabet x at i^{th} position, and the transition is to replace x with y, move right and be in state q', it can be encoded to our rule model in a rule (p, e) defined as follows:

$p_{i,x,q} = in(i, x) \wedge at(i, q)$
$e_{i,x,y,q,q'} = \{\neg in(i, x),\ \neg at(i, q),\ in(i, y),\ at(i+1, q')\}$

Similarly, if the head is at position $i > 0$ and moves left, it can be encoded to our rule model in a rule (p, e) as follows:

$p_{i,x,q} = in(i, x) \wedge at(i, q)$
$e_{i,x,y,q,q'} = \{\neg in(i, x),\ \neg at(i, q),\ in(i, y),\ at(i-1, q')\}$

Here we limit the value of i to $i < n-1$ and $i > 0$, for rules corresponding to right and left movement of the head respectively, to handle boundary conditions.

Now q_{accept} is the accepting state of the Turing machine where it halts. We assume that the Turing machine moves its head all the way back to the 0^{th} position and clears the entire tape when it reaches the accepting state before halting, i.e., $at(0, q_{accept})$ is TRUE and $in(i, \#)$ is TRUE for all possible values of i. To reduce this Turing machine to CONFLUENCE problem, we will add one more transition rule to reach q_{accept} directly from initial configuration where $at(0, q_0)$ is TRUE. We add a new rule (p_{new}, e_{new}) corresponding to the transition.

$p_{new} = at(0, q_0)$
$e_{new} = \{at(0, q_{accept}),\ \neg at(0, q_0),\ in(0, \#),\ in(1, \#), \ldots, in(n-1, \#)\}$
$\cup\ \{\neg in(i, x) \mid 0 \leq i \leq n-1 \text{ and } x \in \Gamma \setminus \{\#\}\}$

This concludes our encoding of a polynomial space bounded Turing machine to our confluence problem. The newly produced rule set will be confluent if the Turing machine M reaches accepting state, it will be non-confluent otherwise. The extra rule (p_{new}, e_{new}) applicable in initial state completes the rule model. If the space bounded Turing machine M accepts an input, after executing its transitions, it will

end in an accepting state. We have corresponding rules in our rule set for each transition through which our rule set will also reach the accepting state. But there is also another path through the new rule (p_{new}, e_{new}). So the rule set is confluent. If the Turing machine M rejects the input, M reaches the rejecting state instead of accepting state, and in our rule set the paths would never merge into single state. So, if an input is rejected, only one path through (p_{new}, e_{new}) reaches q_{reject} and the system is not confluent.

Figure 3 shows a visualization of the reduction. The rule (p_{new}, e_{new}) is a new rule which is added to make system confluent.

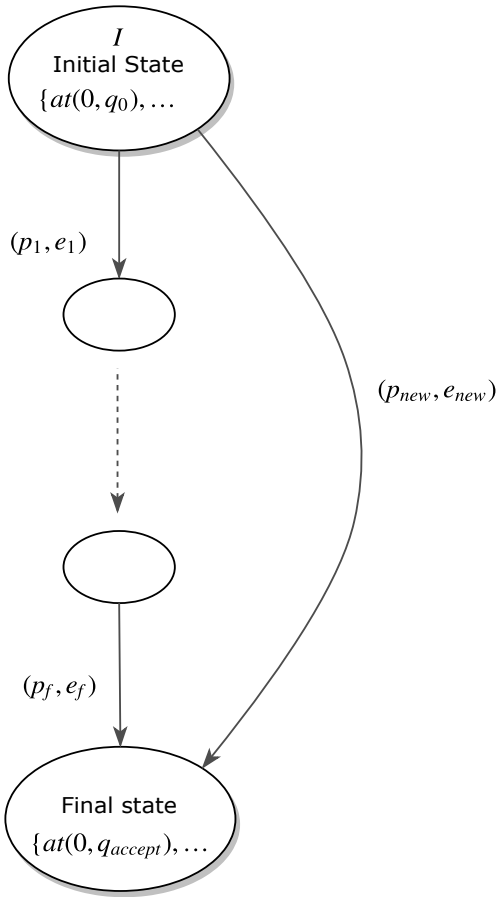

Figure 3: Confluent system

Because the rules can precisely encode the transitions, the system is confluent if and only if the Turing machine halts in an accepting state. The system would not be confluent if the Turing machine does not reach the accepting state. The state variables in our rule model are defined using i, q and x. The values of i depend on the tape length n which is polynomial $f(k)$ in terms of input length k. Also the values which q can take depend on the number of states of the Turing machine which is polynomial in terms of the Turing machine description length $\|M\|$, and x belongs

to the finite alphabet set Γ.

As there are only polynomial number of i, q and x combinations, there can only be polynomial number of rules. As a result, the reduction to CONFLUENCE from PSPACE Turing machine will also be in polynomial time with respect to $f(k)$ and $\|M\|$. As a polynomial space bounded Turing machine can be reduced to CONFLUENCE in polynomial time, CONFLUENCE is PSPACE-hard. □

Theorem 3. *CONFLUENCE is PSPACE-complete.*

From Lemma 1 and Lemma 2, as CONFLUENCE is both in PSPACE and PSPACE-hard, it is PSPACE-complete.

4.3 Termination

Now we focus on the property of termination.

Lemma 4. *A system $(\mathcal{A}, \mathcal{R}, I)$ is terminating if and only if for all states $s_1 \in \mathcal{C}(I)$ and $s_2 \neq s_1$, if $s_2 \in \mathcal{C}(s_1)$, then $s_1 \notin \mathcal{C}(s_2)$.*

Proof. We first prove the implication from left to right. Assume that the system $(\mathcal{A}, \mathcal{R}, I)$ is terminating. If a system is terminating then the system has no infinite execution (from Definition 13). Let $(I, \dots, s_1, \dots, s_2)$ be an execution in the terminating system $(\mathcal{A}, \mathcal{R}, I)$. Here $s_2 \neq s_1$ and $s_2 \in \mathcal{C}(s_1)$. If $s_1 \in \mathcal{C}(s_2)$, then there exists an infinite maximal execution $(I, \dots, s_1, \dots, s_2, \dots, s_1, \dots, s_2, \dots)$ which makes the system non-terminating. So if the system is terminating, $s_1 \notin \mathcal{C}(s_2)$ must hold.

Now we prove the implication from right to left. Assume that in a system $(\mathcal{A}, \mathcal{R}, I)$, for all states $s_1 \in \mathcal{C}(I)$ and $s_2 \neq s_1$, if $s_2 \in \mathcal{C}(s_1)$, then $s_1 \notin \mathcal{C}(s_2)$. Our rule model has a finite number of state variables and therefore finite number of states. Suppose there are t states in total. The maximum number of different states that can occur in an execution is t. If the execution goes on further, there must be two (or more) occurrences of some state s_1. From Definition 7, every rule application must change the state of the system. As a result if the system is non-terminating then there is a maximal execution such as $(I, \dots, s_1, \dots, s_2, \dots, s_1, \dots)$ where some state s_1 is repeated with the system reaching some other state s_2 in between. But this would mean that there is some s_1 and s_2 such that $s_1 \neq s_2$, $s_2 \in \mathcal{C}(s_1)$ and $s_1 \in \mathcal{C}(s_2)$. This contradicts our assumption that $s_1 \notin \mathcal{C}(s_2)$. Therefore the system must be terminating if $s_1 \notin \mathcal{C}(s_2)$. Hence, the lemma holds. □

Lemma 4 basically states that a terminating system cannot have any loops in it. This means that if any state is reachable from another state, the system should not be able to go back to that previous state again. If it can, then the system might get stuck in a loop and may never terminate.

Definition 19 (Instance of the termination problem). An instance of the termination problem is specified by a tuple $(\mathcal{A}, \mathcal{R}, I)$ where $\mathcal{A}$ is a finite set of state variables, $\mathcal{R}$ is a finite set of rules, and I is the initial state given by the valuation of the state variables in $\mathcal{A}$.

We define TERMINATION as the decision problem of determining if an instance $(\mathcal{A}, \mathcal{R}, I)$ of termination problem is a terminating system according to Definition 13 and Lemma 4.

Lemma 5. *TERMINATION is in PSPACE.*

Proof Idea. We can test TERMINATION by checking if a loop can be formed between any two states reachable from the initial state. We can use the reachability algorithm which has been proven to be in PSPACE [8] to prove that TERMINATION is also in PSPACE.

Proof. The reachability problem of finding if a state s_2 is reachable from state s_1, i.e., check if $s_2 \in \mathcal{C}(s_1)$, is in PSPACE. We use the same Algorithm 1: REACH(s_1, s_2, t) described previously to prove that TERMINATION is in PSPACE. For n state variables in the system, the total number of states is 2^n. The set of all possible states of the system is given by $\mathcal{S}$.

Algorithm 3: TERM$(\mathcal{A}, \mathcal{R}, I)$ tests termination for a system $(\mathcal{A}, \mathcal{R}, I)$. For all the states in the system, the algorithm checks if it is reachable from the initial state I. If it is reachable, the algorithm checks every other state to see if a loop can be formed. If a loop can be formed, the system is non-terminating, else it is terminating (from Lemma 4).

Each *for* loop iteration of states for s_1 and s_2 takes polynomial $f(n)$ space if there are n state variables. Also each call to REACH takes polynomial space during its call and returns TRUE or FALSE upon completion. As each part of the algorithm to test TERMINATION takes only polynomial space, the problem is in PSPACE. □

Algorithm 3: TERM($\mathcal{A}, \mathcal{R}, I$)

```
for all states s_1 in S do
    if REACH (I, s_1, 2^n) then
        for all states s_2 in S do
            if s_1 ≠ s_2 AND REACH (s_1, s_2, 2^n) AND REACH (s_2, s_1, 2^n) then
                return FALSE;
            end
        end
    end
end
return TRUE;
```

Lemma 6. *TERMINATION is PSPACE-hard.*

Proof Idea. To prove that TERMINATION is PSPACE-hard, we need to reduce any polynomial space bounded Turing machine to our problem of TERMINATION in polynomial time. We can do so by encoding transitions of the Turing machine to our rule model in a way that it is terminating only when the Turing machine accepts. This can be achieved by introducing an additional rule which forms a loop between the rejecting state of the Turing machine and some other reachable state.

Proof. To prove TERMINATION is PSPACE-hard, we will use the same polynomial space bounded deterministic Turing machine M, defined in the previous proof, which decides a language in PSPACE. We will also use the same procedure as in the previous proof to encode the transitions of the Turing machine to our model.

For M, let n be the length of the tape. So position of the head can be at any place $0, 1, \ldots, n-1$. If the length of input string to the Turing machine is k, n is a polynomial function $f(k)$. We define state variables as

- $in(i, x)$ = symbol x is in tape position i, $x \in \Gamma$, and
- $at(i, q)$ = the Turing machine is in state q and the head is in position i.

So if the Turing machine contains input on the tape from position 1 to k, the initial configuration of the Turing machine can be encoded in our rule model as,

$I = at(0, q_0),\ in(0, \#),\ in(1, x_1),\ in(2, x_2), \ldots,\ in(k, x_k),$
$in(k+1, \#),\ in(k+2, \#), \ldots, in(n-1, \#)$ variables are TRUE
... and all other state variable are FALSE.

The rules $(p_{i,q,x}, e_{i,q,x})$ are also defined similarly to the previous proof. For Turing machine in state q, the machine head at position $i < n-1$, and the head reading alphabet x at i^{th} position, the transition to replace x with y, move right, and be in

state q' can be encoded to our rule model in a rule (p, e) defined as follows:

$$p_{i,x,q} = in(i,x) \wedge at(i,q)$$
$$e_{i,x,y,q,q'} = \{\neg in(i,x),\ \neg at(i,q),\ in(i,y),\ at(i+1,q')\}$$

We can handle all transitions of the Turing machines using such rules as discussed in the proof of Lemma 2. We limit the value of i to $i < n-1$ and $i > 0$, for rules corresponding to right and left movement of the head respectively, to handle boundary conditions.

Now, instead of q_{accept} we will add a rule in our model from q_{reject}. The state q_{reject} is the rejecting state of the Turing machine where it halts while rejecting the input. We assume that the Turing machine moves its head all the way back to the 0^{th} position and clears the entire tape when it reaches the rejecting state before halting, i.e., $at(0, q_{reject})$ is TRUE and $in(i, \#)$ is TRUE for all possible values of i. To reduce this Turing machine to TERMINATION problem, we will add one more transition rule from the rejecting state q_{reject} to the initial configuration where $at(0, q_0)$ is TRUE. We add a new rule (p_{new}, e_{new}) corresponding to the transition.

$$\begin{aligned} p_{new} &= at(0, q_{reject}) \\ e_{new} &= \{at(0,q_0), \neg at(0,q_{reject})\} \\ &\quad \cup \{in(i,x) \mid 0 \le i \le n-1, x \in \Gamma \text{ and } in(i,x) \text{ is TRUE in state } I\} \\ &\quad \cup \{\neg in(i,x) \mid 0 \le i \le n-1, x \in \Gamma \text{ and } in(i,x) \text{ is FALSE in state } I\} \end{aligned}$$

This rule makes the system go back to the state I corresponding to the initial configuration.

This concludes our encoding of a polynomial space bounded Turing Machine to our TERMINATION problem. The newly produced rule set will be terminating if the Turing machine M reaches accepting state, it will form a loop if it rejects. If M accepts an input in PSPACE, after executing its transitions, it will end in an accepting state. We have corresponding rules in our rule set for each transition through which our rule set will reach the accepting state. As there are no loops in this path, the system is terminating. But if it reaches a rejecting state, the new rule (p_{new}, e_{new}) ensures that there is a path back to initial state and the system is non-terminating.

Figure 4 shows a visualization of the reduction. The rule (p_{new}, e_{new}) is a new rule which takes the system from the rejecting state to I. This forms a loop and the system would be non-terminating. If it reaches accepting state, the system would terminate.

Because the rules can precisely encode the transitions, the system is terminating if and only if the Turing machine halts in an accepting state. The system would not be terminating if the Turing machine does not reach the accepting state. The

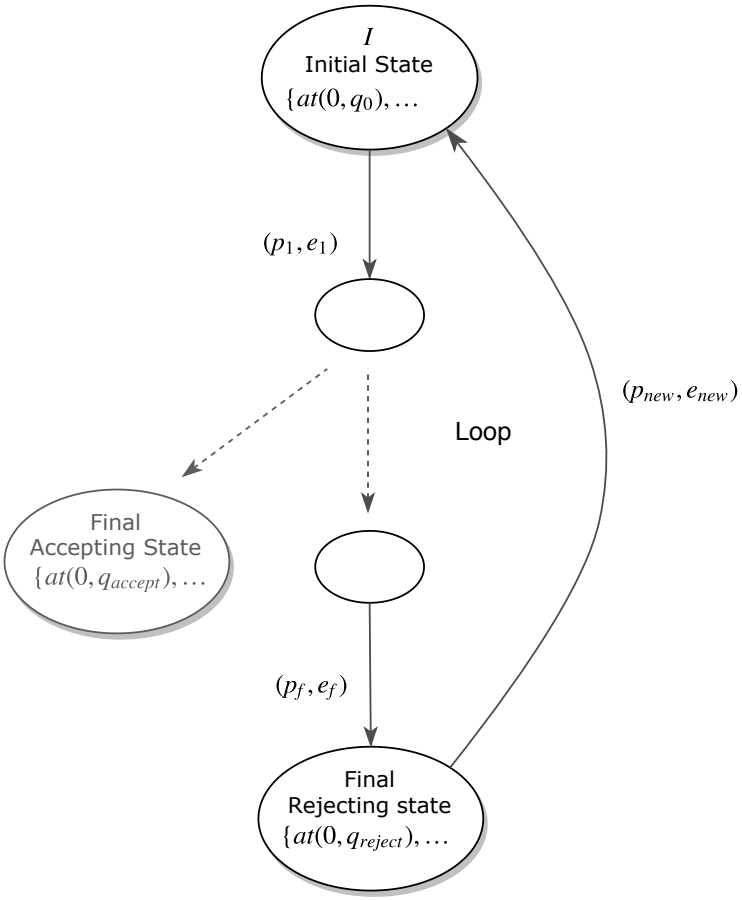

Figure 4: Non-terminating system

state variables in our rule model are defined using i, q and x. The values of i depend on the tape length n which is polynomial $f(k)$ in terms of input length k. Also the values which q can take depend on the number of states of the Turing machine which is polynomial in terms of the Turing machine description length $\|M\|$, and x belongs to the finite alphabet set Γ.

As there are only polynomial number of i, q and x combinations, there can only be polynomial number of rules. As a result, the reduction to TERMINATION from PSPACE Turing machine will also be in polynomial time with respect to $f(k)$ and $\|M\|$. As a polynomial space bounded Turing machine can be reduced to TERMINATION in polynomial time, TERMINATION is PSPACE-hard. □

Theorem 7. *TERMINATION is PSPACE-complete.*

From Lemma 5 and Lemma 6, as Termination is both in PSPACE and PSPACE-hard, it is PSPACE-complete.

4.4 Boundedness

A process needs computing resources in the form of time and space when carrying out instructions. A system might be terminating but may have a lot of rules to fire in a single execution sequence and take quite a long time to complete it. For n Boolean state variables, the total number of states in the system is 2^n. This is the maximum number of states that a terminating system can reach in a single execution. The important thing to note here is that this upper bound is exponential with respect to number of state variables. Therefore there is an obvious practical interest to test, whether the length of the rule sequences is bounded more strongly than this. Finding the minimum bound on the length of executions is useful while designing a system using state transition models.

Let us analyse the relationship of the bound with the number of rules in the system. Let x be the number of rules in the transition system $(\mathcal{A}, \mathcal{R}, I)$. If we limit the length of the executions to a constant c, then there can be at most x^c different transition sequences of length c. This is polynomial in x. We can test this bound by checking that there are no rules applicable in all states reachable by a transition sequence of length c. Hence the test for boundedness for constant length bounds is polynomial.

Table 1: Complexity results for different bounds

Bound	Complexity
c	P
$p(x)$	NP
$2^{p(x)}$	PSPACE

If the bound is polynomial in x, then there are at most $x^{p(x)}$ transition sequences to check. This is not polynomial any more. However, we can use a non-deterministic polynomial time process for testing boundedness as follows: Guess a sequence of $p(x)$ actions non-deterministically, and if the state reachable has applicable rules in it, then the system is not bounded, otherwise it is bounded. As the *guessing* part is done non-deterministically, the problem of determining if the system is bounded with some $p(x)$ can be solved by a non-deterministic Turing machine. Hence the problem of testing boundedness for polynomial bounds is in NP.

For an exponential bound, the total number of transitions can be at most $x^{2^{p(x)}}$. In this case the complexity of finding the bound is PSPACE as we can go through each execution one by one testing its length with only the current state saved in memory, which requires polynomial amount of space.

5 Practical Considerations

5.1 Rule Model with Event Rules

In the previous sections, we analysed confluence and termination for executions starting from the given initial state. When a system terminates after a maximal execution, we can say that the system has stabilized as no rules are applicable. So we call this a stable state. Now in real world usually some event occurs, e.g., some internal change in the system or a user action, which triggers some rules again. We use special kind of rules called *event rules* to model these events. The event rules makes other non-event rules applicable. Whenever rules other than event rules are applicable, the system always runs for a maximal execution until it terminates. In other words, these other rules are *forced* to be fired for maximal execution, so we call non-event rules as *forced rules*.

The new rule model with events is represented as $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$ where $\mathcal{A}$ is a finite set of state variables, $\mathcal{R}$ is a finite set of forced rules, $\mathcal{U}$ is a finite set of event rules, and I is the initial state. Now our rule model consists of two types of rules: event rules and forced rules. The difference between them is that the event rules may or may not fire even if they are applicable, whereas one of the applicable forced rules must always fire until none of the forced rules remain applicable. The event rule is not forced as we use it to model something not in control of the system itself. We consider an event happening as a consequence of a stable state for the purpose of analysis, and we consider stable state as a state in which none of the forced rules are applicable.

Definition 20 (Stable state). State s is called a *stable state* if either of the following holds for all forced rules $(p, e) \in \mathcal{R}$.

1. $s \not\models p$.
2. $succ_e(s) = s$.

This means that for a state to be stable, for all forced rules in the rule model, either their precondition is FALSE or their effects would not change the state. For a terminating maximal execution, the final state in which the execution terminates is a stable state as none of the forced rules are applicable. Before an execution starts again, there has to be a change in the system state which takes it from a stable state to a state where rules are applicable again. This is an event indicated by an event rule in our rule model.

The application of event rules is only considered in stable states. The definition of an event rule is syntactically same as the definition of a rule. So an event rule can be thought of as a special kind of a rule. For the transition system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$,

the event rules in $\mathcal{U}$ determine how the new executions will start after the system has already completed its first execution. Whenever an event occurs, the system responds to it by firing applicable forced rules to maximal execution. Also only one event can occur until the system terminates and not all events can occur in all stable states. An event can occur in a stable state only if it is applicable.

Because we differentiate between event rules and forced rules, we also slightly modify the definition of reachability and maximal execution to fit this modified rule model.

Definition 21 (Reachable state in a rule model with events). A state s' is reachable from s if and only if there is an execution $(s, \ldots, s')$ using only **forced rules** from state s to state s'.

The set of all the states reachable from s is denoted by $\mathcal{C}(s)$. So, if s' is reachable from s using only forced rules, we denote it as $s' \in \mathcal{C}(s)$.

Definition 22 (Maximal execution for a system with events). An execution is *maximal* for a system with events if either of the following holds.

1. The execution using only forced rules is finite, hence of the form $(s_1, s_2, \ldots, s_n)$, and no forced rule is applicable in state s_n.
2. The execution using only forced rules is infinite.

In practical applications usually there are many states that the system can never be in. For example, if we represent the motion of a robot using a transition system, where different Boolean state variables correspond to the position of the robot, then these variables can take only limited values due the the constraint that the robot cannot be in multiple positions at once. The states of the system which represent that the robot is in two places at once are intrinsically not attainable. We need to eliminate such states from the system when we analyse properties such as confluence and termination. We call stable states that the system can actually reach at some point using both forced rules and event rules as *s-reachable states*.

Definition 23 (S-reachable state). For a transition system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$, we define $\mathcal{S}_t$, a set of *s-reachable* states, as follows:

1. If $s \in \mathcal{C}(I)$ is a stable state, then $s \in \mathcal{S}_t$.
2. If s is an s-reachable state, then any stable state $s' \in \mathcal{C}(succ_e(s))$, for any event rule $(p, e) \in \mathcal{U}$ applicable in s, is also an s-reachable state.
3. The set being defined is the smallest set that satisfies (1) and (2).

S-reachable states are stable states and no forced rules are applicable in these states. Finding s-reachable states is quite useful as we can focus on analysing events in only s-reachable states and safely ignore other stable states that can never be reached. The executions begin after an event occurs in an s-reachable state and the system moves to its successor state where the forced rules could become applicable. We call the successor states of s-reachable states due to any event rule as *e-reachable states*.

Definition 24 (E-reachable state). For a transition system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$, we define $\mathcal{S}_e$, a set of *e-reachable* states, as follows:

1. Initial state $I \in \mathcal{S}_e$.
2. For any s-reachable state s and any event rule $(p, e) \in \mathcal{U}$ applicable in s, if $s' = succ_e(s)$, then $s' \in \mathcal{S}_e$.

We include initial state I in the set $\mathcal{S}_e$ of e-reachable states as it is also a state from where executions using forced rules could begin. Note that a state can be both an s-reachable state and an e-reachable state if no forced rule is applicable in the state and it is also a successor of some other s-reachable state due to some event rule.

Successor of a state with respect to a rule is defined exactly the same as before. One important thing to note here is that successor to a stable state is considered only from an event rule, while for all other states the successor is considered only from forced rules. The set of successor states of state s is denoted by $\mathcal{P}(s)$.

Now we look at the properties we are interested in studying for a rule model with events.

Definition 25 (Confluent system with events). A system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$ is *confluent* if for all $s_1, s_2 \in \mathcal{C}(s)$ where s is an e-reachable state, there exists a state s_3 such that $s_3 \in \mathcal{C}(s_1)$ and $s_3 \in \mathcal{C}(s_2)$.

Definition 26 (Terminating system with events). A system with events is *terminating* if there is no infinite execution sequence using only forced rules in the system starting from an e-reachable state, or in other words, all maximal executions starting from an e-reachable state using only forced rules are finite.

Definition 27 (Bound of a terminating system with events). A terminating system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$ is called N-bounded, if for all e-reachable states s there is no execution $(s, s_1, ..., s_n)$ using only forced rules such that $n > N$.

Under this modified rule model with events, the first execution always starts in the initial state I. We have already analysed such executions from starting state I in the

previous section, so we move our focus to other executions. All other executions in the rule model with events start with an s-reachable state s and an event $(p_v, e_v) \in \mathcal{U}$. This takes the system to an e-reachable state $s' = succ_{e_v}(s)$ where forced rules $(p_f, e_f) \in \mathcal{R}$ could become applicable. If they are applicable, firing these rules can make other rules applicable leading to potentially longer sequences of rule firings and this continues until a stable state is reached again. One important thing to note is that after an event occurs, another event cannot occur again before the system has terminated. As the system has maximal execution, all applicable forced rules are *forced* to fire until a stable state is reached and only then can another event occur if it is applicable in that stable state. As a result the set of reachable states for any execution depends not only on the forced rules but also on the e-reachable state the execution starts in. E-reachable states in turn depend on the s-reachable states and the events which occur in those states. Different s-reachable states with same event may lead to different e-reachable states, while different events from the same s-reachable state may also lead to different e-reachable states. The later is shown in Figure 5. The figure shows only a part of the state transition graph where different events from the same s-reachable state lead to different e-reachable states and different terminating states.

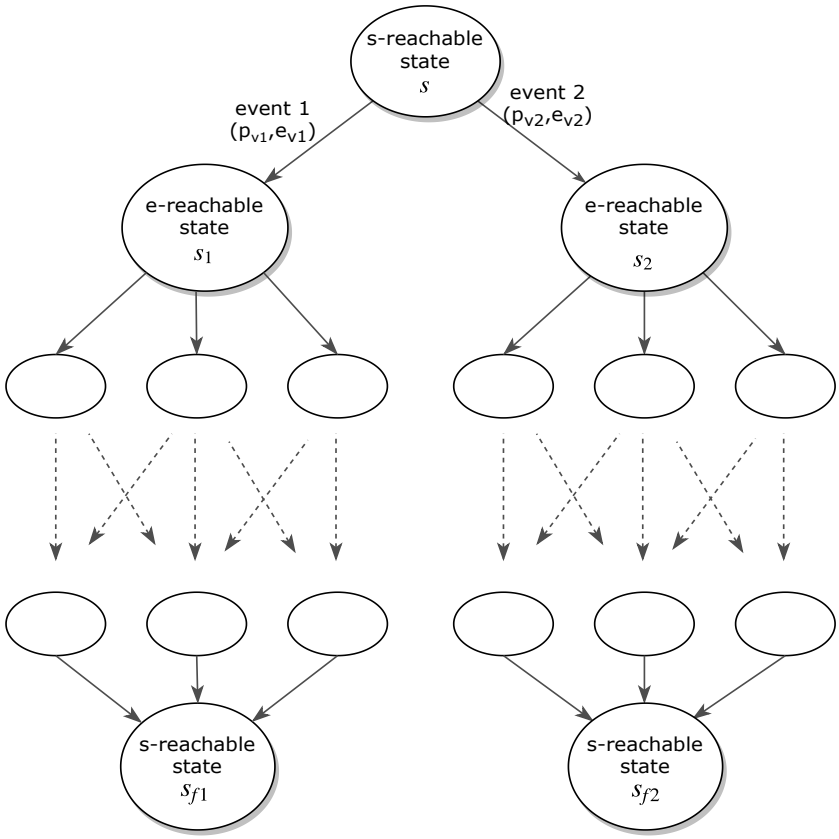

Figure 5: Termination and confluence with different events

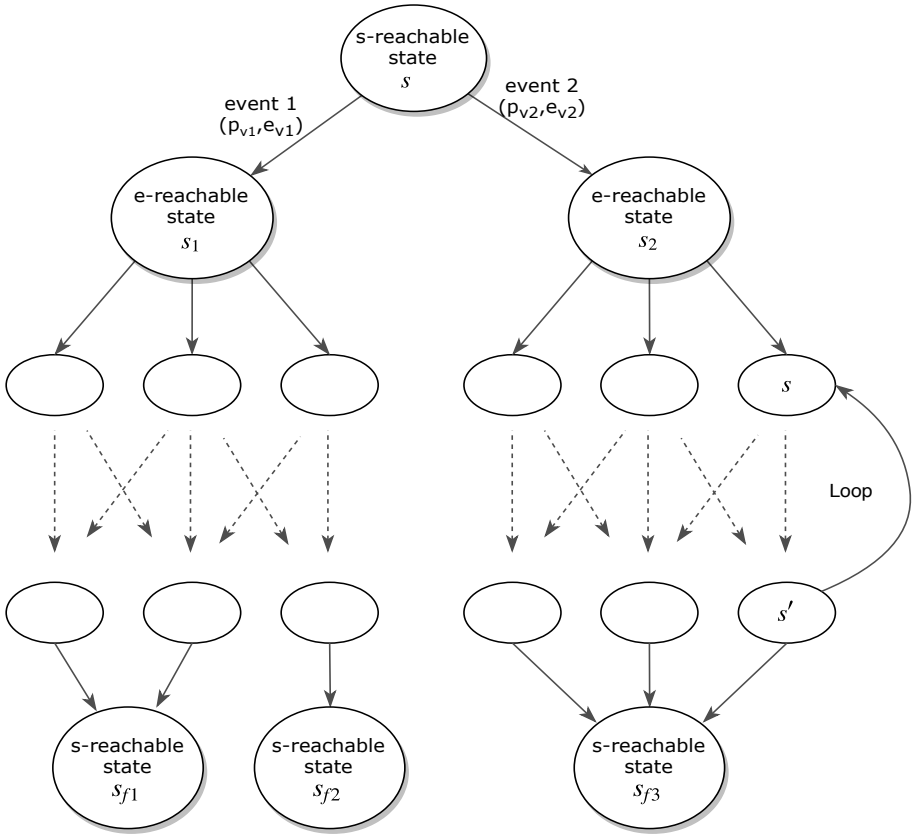

Figure 6: Non-terminating and non-confluent system with events

In a confluent system, executions starting from the initial state I are confluent. For a system with events to be confluent, the executions from all possible e-reachable states should also be confluent. If for any e-reachable state, the execution is not confluent, the entire system is called non-confluent. Similarly, for a system with events to be terminating, there should be no loop in any execution branch from any of the e-reachable states. Even if there is a loop formed on an execution path of only one of the e-reachable states, the system is non-terminating. A part of state-transition graph for a non-confluent and non-terminating system is shown in Figure 6. The execution through event 1 is terminating but not confluent, so the system is not confluent. The execution through event 2 is non-terminating. As there is one non-terminating execution branch, we say that the system is non-terminating.

5.2 Analysis of Rule Model with Events

We now see how the introduction of events in our rule model changes the complexity of termination and confluence problems.

5.2.1 Confluence and Termination

Instances for CONFLUENCE and TERMINATION decision problems for the modified rule model are now given by $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$ where $\mathcal{A}$ is a finite set of state variables, $\mathcal{R}$ is a finite set of forced rules, $\mathcal{U}$ is a finite set of event rules, and I is the initial state.

We have already analysed executions starting from the initial state I in the previous section. So now we analyse executions starting from some e-reachable state. Before we do this, we need to find all s-reachable states in the system. Once we do that we can use events to find the e-reachable states. Algorithm 4: SREACH(s_1, s_2, t) is similar to Algorithm 1: REACH(s_1, s_2, t), but SREACH(s_1, s_2, t) also takes into account the stable states and uses event rules to find successors of those states. The call SREACH(s_1, s_2, t) tests if s_2 is reachable in t steps from s_1 using both forced rules and event rules if no forced rules are applicable. The number of steps t is in powers of 2 here and the maximum value it can take is 2^n as it is the maximum number of states the system can have for n Boolean state variables. The function calls itself recursively, and each level has to store only the two states and the value of t. Furthermore, the value of t is also halved in each recursion. So the space needed by the algorithm is $\mathcal{O}(\log t)$, or in other words $\mathcal{O}(f(n))$ which is polynomial in n.

Algorithm 4: SREACH(s_1, s_2, t)

```
if t = 1 then
    if s1 = s2 then
        return TRUE;
    else if s2 = succ_e(s1) for some forced rule (p, e) ∈ R applicable in s1
     then
        return TRUE;
    else if no forced rule is applicable in s1 AND s2 = succ_e(s1) for some
     event rule (p, e) ∈ U applicable in s1 then
        return TRUE;
    else
        return FALSE;
    end
else
    for all states s' do
        /* Use recursion                                              */
        if SREACH (s1, s', t/2) AND SREACH (s', s2, t/2) then
            return TRUE;
        end
    end
    return FALSE;
end
```

Algorithm 5 defines a function isSreachable$(s, \mathcal{R})$ which takes a state as input and checks first if it is stable using all the forced rules in the system and then checks if it

is s-reachable from initial state I using Algorithm 4: SREACH(s_1, s_2, t). This is also in PSPACE as there are only finite number of forced rules and as SREACH(s_1, s_2, t) runs in PSPACE. In this algorithm, n is the number of state variables.

Algorithm 5: isSreachable(s)

```
for all (p, e) ∈ R do
    /* Check if precondition is true for any forced rule and if
       effect changes the state.                                  */
    if s ⊨ p AND succ_e(s) ≠ s then
        return FALSE;
    end
end
/* Check if s-reachable.                                         */
if SREACH (I, s, 2^n) then
    return TRUE;
end
return FALSE;
```

We can use Algorithm 5 to find an s-reachable state and then iterate through all the event rules $(p, e) \in \mathcal{U}$ that can be applied in this state. The successor state $succ_e(s)$ is an e-reachable state obtained on firing an event rule (p, e) in s-reachable state s. From this state onwards only the forced rules in the rule model are relevant. We have already proven in the previous section that testing confluence for a system with a rule set $\mathcal{R}$ and no events is in PSPACE (from Lemma 1). We use the Algorithm 2: CONF$(\mathcal{A}, \mathcal{R}, succ_e(s))$ by using $succ_e(s)$ to define a transition system in place of I. This is shown in Algorithm 6. Here $\mathcal{S}$ is the set of all states in the system. As Algorithm 2 runs in polynomial space, and as finding all e-reachable states along with iterating through finite number of states and events runs in PSPACE, we can say that testing confluence for a rule model with events is in PSPACE.

Algorithm 6: CONF_EVENT $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$

```
for all s ∈ S do
    if isSreachable(s) then
        for all (p, e) ∈ U applicable in s do
            if CONF(A, R, succ_e(s))= FALSE then
                return FALSE;
            end
        end
    end
end
return TRUE;
```

Similarly, for termination problem, we use Algorithm 3: TERM$(\mathcal{A}, \mathcal{R}, succ_{e_v}(s))$ which runs in PSPACE, to test termination as shown here in Algorithm 7. Also

iterating through all possible s-reachable state and event combination needs polynomial space. Hence termination problem for rule model with events is also in PSPACE.

Algorithm 7: TERM_EVENT $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$

```
for all s ∈ S do
    if isSreachable(s) then
        for all (p, e) ∈ U applicable in s do
            if TERM(A, R, succ_e(s))= FALSE then
                return FALSE;
            end
        end
    end
end
return TRUE;
```

We have already proven that CONFLUENCE and TERMINATION are PSPACE-hard for transition system $(\mathcal{A}, \mathcal{R}, I)$ without events. We can represent the same transition system as a special case of transition system with events $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$ where $\mathcal{U} = \emptyset$. As every transition system without events $(\mathcal{A}, \mathcal{R}, I)$ can be reduced to a transition system with events $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$, testing confluence and termination for the later is at least as hard as the former. We can also directly reduce the space bounded Turing machine to the special case of transition system with empty event rule set exactly as we did in the proofs of Lemma 2 and Lemma 6. Hence, testing confluence and termination for a rule model with events is PSPACE-hard.

Corollary 1 (to Theorem 3)**.** *CONFLUENCE for a rule model with events is PSPACE-complete.*

Corollary 2 (to Theorem 7)**.** *TERMINATION for a rule model with events is PSPACE-complete.*

As solving CONFLUENCE and TERMINATION for a rule model with events is both in PSPACE and PSPACE-hard, it is PSPACE-complete. We see here that introduction of events does not change the space complexity of solving these problems.

5.2.2 Boundedness

Analysing boundedness in terms of number of state variable is exponential for a rule model without events, and it also holds true for a rule model with events as the number of states is still exponential in terms of state variables. If we analyse the relationship of the bound with the number of rules and events in the system, we quickly see that finding the bound is much harder with events than the system

without events. Even if we limit ourselves to a constant bound c on the length of executions, we still need to find all e-reachable states first and then test the length of executions starting from those states. Finding all s-reachable states and subsequently all e-reachable states is in PSPACE as shown by the Algorithm 5. This also holds for polynomial bounds. So testing boundedness for a system with events is in PSPACE unlike the system without events for which it was in P and NP for constant and polynomial bounds respectively.

6 Restricted And Tractable Cases

As seen in the previous sections, the confluence and termination problems are PSPACE-complete and therefore intractable. In this section we look at some special cases or scenarios, in which the problems might become more tractable. We are interested in algorithms which run in polynomial time and can be used in practical applications.

6.1 Rule Model with Non-Interfering Rules

Depending on how the rules are defined, the confluence and termination problems might be solved in polynomial time in some cases. For this we will look at interference among rules first. A forced rule interferes with another forced rule, if the literals in its effect can falsify the precondition or negate any literal in the effect of the other forced rule. The concept of interference has first been discussed by Blum and Furst [7] to avoid considering actions in many different orders, to speed up search for planning problem. In their framework several actions may be specified to occur at the same time step of the plan so long as they do not interfere with each other.

Definition 28 (Positive and negative occurrences). We define the positive and negative occurrences of a state variable a in a propositional formula ϕ over the set $\mathcal{A}$ of state variables follows:

1. a occurs positively in a, for all $a \in \mathcal{A}$,
2. a occurs positively in $\phi \wedge \phi'$ if it occurs positively in ϕ or ϕ',
3. a occurs positively in $\phi \vee \phi'$ if it occurs positively in ϕ or ϕ',
4. a occurs positively in $\neg\phi$ if it occurs negatively in ϕ,
5. a occurs negatively in $\phi \wedge \phi'$ if it occurs negatively in ϕ or ϕ',
6. a occurs negatively in $\phi \vee \phi'$ if it occurs negatively in ϕ or ϕ',
7. a occurs negatively in $\neg\phi$ if it occurs positively in ϕ.

A state variable a occurs in ϕ if it occurs positively or occurs negatively in ϕ.

Definition 29 (Interference). A forced rule $(p_1, e_1) \in \mathcal{R}$ *interferes* with forced rule $(p_2, e_2) \in \mathcal{R}$ if any of the following holds.

1. $e_1 \cap \{\bar{l} \mid l \in e_2\} \neq \emptyset$

2. $e_2 \cap \{\bar{l} \mid l \in e_1\} \neq \emptyset$
3. For some $a \in e_1$, a occurs in p_2 negatively.
4. For some $\neg a \in e_1$, a occurs in p_2 positively.
5. For some $a \in e_2$, a occurs in p_1 negatively.
6. For some $\neg a \in e_2$, a occurs in p_1 positively.

If (p_1, e_1) interferes with (p_2, e_2), it logically follows that (p_2, e_2) also interferes with (p_1, e_1). So they are both a pair of interfering rules.

Definition 30 (Non-interfering rules). If two forced rules (p_1, e_1) and (p_2, e_2) do not interfere with each other, they are *non-interfering* with respect to each other. If a forced rule does not interfere with any other forced rule in the system, the rule is a *non-interfering rule.*

Note. We use the term non-interfering rule to mean that a forced rule is non-interfering with all other forced rules in the rule model unless it is specifically mentioned that the rule is non-interfering with some other specific rule.

In our rule model, non-interference is relevant only to the forced rules as the final state of the execution depends on what order the forced rules are fired. For event rules, only one event rule is fired at the start of an execution and other event rules do not affect the executions starting from that particular event.

The effects of a non-interfering rule do not falsify the precondition of any other forced rule and also do not negate the effects of any other forced rule in the rule set. But more importantly, no other forced rule can falsify the precondition of the non-interfering rule or negate its effects. After any forced rule from the rule set is fired, the precondition of a non-interfering rule can never become FALSE if it was TRUE before the other rule fired. Also once the non-interfering rule has fired, the literals in the effect will stay TRUE until the system terminates as no other forced rule can negate it.

Lemma 8. *If a non-interfering rule (p, e) is fired in an execution and the final state of the execution is s_f, then $s_f \models e$. If multiple non-interfering rules $(p_1, e_1), (p_2, e_2), \ldots, (p_k, e_k)$ are fired in an execution and the final state of the execution is s_f, then $s_f \models E$ where $E = e_1 \cup e_2 \cup \cdots \cup e_k$.*

Proof. If one non-interfering rule (p, e) is fired at any point in an execution sequence, then by definition of non-interfering rules, the literals in its effect are never falsified

as none of the other forced rules interfere with it. So if the resulting state of the execution is s_f, $s_f \models e$. Similarly, for multiple non-interfering rules, none of the literals in their effects are falsified (by definition) by other forced rules. As a result, if any number of non-interfering rules $(p_1, e_1), (p_2, e_2), \ldots, (p_k, e_k)$ are fired at any position in an execution, each literal in all their effects still hold in the resulting state, i.e., $s_f \models E$ where $E = e_1 \cup e_2 \cup \cdots \cup e_k$. □

Theorem 9. *If all forced rules in the system are non-interfering, then the system is terminating.*

Proof. Consider any execution sequence in the system, and let $(p_1, e_1), (p_2, e_2), \ldots, (p_n, e_n)$ be the rules that are fired at least once in the execution sequence. Suppose one of these rules (p, e) is applicable and fired in some state s_i of the execution. So all effects of the rule (p, e) are TRUE in the successor state, that is, if successor state to s_i after firing rule (p, e) is s_{i+1}, then $s_{i+1} \models e$. Since the rule set is non-interfering, none of the rules fired later can make any literal in e false. So for the next states in the execution, $s_{i+2} \models e$, $s_{i+3} \models e$, and so on. As e will always hold in all successive states, the rule (p, e) cannot become applicable again, and therefore there can be no second occurrence of (p, e) in the execution. As this holds for any (p, e) in the execution, there can be at most one occurrence of any rule. Also the total number of rules in the rule model is finite. Therefore the all executions must be finite and the system is terminating. □

Lemma 10. *If the forced rule set $\mathcal{R}$ in a transition system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$ has only non-interfering rules, then all maximal execution sequences starting from the same e-reachable state have the same set of resulting effects.*

Proof. Let the system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$ under consideration have all non-interfering rules in $\mathcal{R}$. From Theorem 9, we know that if all forced rules in the rule set are non-interfering, the system is terminating. So each maximal execution is terminating. Let $\mathcal{M} = \{(p_1, e_1), (p_2, e_2), \ldots, (p_m, e_m)\}$ be an ordered set of forced rules fired in some maximal execution sequence starting from an e-reachable state s_e and with final state s_f. Using Lemma 8, we can say that $s_f \models E$ where $E = e_1 \cup e_2 \cup \cdots \cup e_m$.

Let $\mathcal{M}' = \{(p'_1, e'_1), (p'_2, e'_2), \ldots, (p'_n, e'_n)\}$ be an ordered set of forced rules fired in some other maximal execution starting from the same e-reachable state s_e and having final state s'_f. Using Lemma 8, $s'_f \models E'$ where $E' = e'_1 \cup e'_2 \cup \cdots \cup e'_n$.

First we use induction to show that $E \subseteq E'$. We show that for every rule $(p, e) \in \mathcal{M}$, either $(p, e) \in \mathcal{M}'$, or some other rules in $\mathcal{M}'$ have the same set of effects as the rule (p, e). The mathematical induction is on i for the ordered rules $\{(p_1, e_1), (p_2, e_2), \ldots, (p_i, e_i)\} \in \mathcal{M}$.

Induction hypothesis: Each rule in the ordered set $\mathcal{M}$ until the rule (p_i, e_i) is either fired in $\mathcal{M}'$ or the literals in its effect are made true by other rules in $\mathcal{M}'$.

Base case $i = 1$: For first rule (p_1, e_1) in the execution sequence $\mathcal{M}$, $s_e \models p_1$. As all forced rules in the system are non-interfering, no other rule can falsify the pre-condition of (p_1, e_1). As a result p_1 will always be TRUE as it is applicable in the starting state s_e. After precondition has been made TRUE, in a maximal execution, the only way a non-interfering rule can stop being applicable is if the literals in its effect have also been made TRUE. So either the rule (p_1, e_1) has to be fired or its effects have to become TRUE due to some other rules in any execution starting from s_e. Hence for any other maximal execution $\mathcal{M}'$, either $(p_1, e_1) \in \mathcal{M}'$ or some other rules in $\mathcal{M}'$ have the same set of effects as the rule (p_1, e_1).

Inductive step $i \geq 1$: From the induction hypothesis, for each rule $(p, e) \in \mathcal{M}$ until the rule $(p_i, e_i) \in \mathcal{M}$, either it is fired in $\mathcal{M}'$ or the literals in its effect are made true by other rules in $\mathcal{M}'$. For $(p_{i+1}, e_{i+1}) \in \mathcal{M}$, the precondition p_{i+1} holds after the effects of all rules till (p_i, e_i) are made TRUE. As all forced rules in the system are non-interfering, no other rule can falsify the pre-condition of (p_{i+1}, e_{i+1}). So either the rule (p_{i+1}, e_{i+1}) has to be fired or literals in its effect have to be made TRUE by some other rules in execution sequence $\mathcal{M}'$.

As a result for every rule $(p, e) \in \mathcal{M}$, either $(p, e) \in \mathcal{M}'$, or some other rules have the same set of effects as the rule (p, e). Also the effects of the fired rules cannot be falsified until the maximal execution terminates as the rules are non-interfering. Hence we can say $E \subseteq E'$. We can similarly prove that every rule in $\mathcal{M}'$ is also present in $\mathcal{M}$ or its effects are made true by other rules in $\mathcal{M}$, showing $E' \subseteq E$. Thus, we can say $E = E'$. Also as this would hold for all maximal executions, we can say that all maximal execution sequences starting from the same e-reachable state have the same set of resulting effects. □

Theorem 11. *If the forced rule set $\mathcal{R}$ in a transition system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$ has only non-interfering rules, then the system is confluent.*

Proof. From Theorem 9, we know that if all forced rules in the rule set are non-interfering, the system is terminating. For the system to be confluent, the terminating state needs to be the same for all maximal executions starting from the same e-reachable state.

For a transition system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$, let s_e be any e-reachable state from where an execution starts. From Lemma 10, we know that all maximal executions starting from the same e-reachable state have same set of effects. Let this set of effects be denoted by E. So if a maximal execution starting from an e-reachable state s_e has a final state s_f, then

1. $s_f \models E$, and

2. $s_f(a) = s_e(a)$ for all $a \in \mathcal{A}$ that do not occur in E.

As all maximal executions have the same set of effects, the final state for all maximal executions starting from s_e is the same state s_f. As all executions starting from any e-reachable state are confluent, the system is confluent. □

Theorem 12. *Testing if all forced rules of the system are non-interfering is polynomial time.*

Proof. Algorithm 8 tests if two rules are interfering in PTIME. It checks all the conditions stated in Definition 29. Both rules should not have any contradicting effects and none of the rules should be able to falsify the precondition of the other. This involves testing each literal from one rule against each literal and state variable in the effect and precondition of the other rule.

To check if a literal **can falsify** a precondition, we need to check the following:

1. If the variable of the positive literal occurs negatively in the precondition of the other rule.

2. If the variable of the negative literal occurs positively in the precondition of the other rule.

The maximum number of literals in the rule are limited by the maximum number of state variables. So if there are n state variables, the worst case time complexity of testing interference is $\mathcal{O}(n^2)$ which is PTIME.

Algorithm 8: interferes$((p_1, e_1), (p_2, e_2))$

```
for all l ∈ e1 do
    if l̄ ∈ e2  OR  l can falsify p2 then
        return TRUE;
    end
end
for all l ∈ e2 do
    if l can falsify p1 then
        return TRUE;
    end
end
return FALSE;
```

Algorithm 9 iterates over all pairs of forced rules in the rule set and tests interference for each pair. If there are m rules, iterating over all rules has time complexity $\mathcal{O}(m^2)$ which is also PTIME. So testing if all rules in the system are non-interfering is

polynomial time. □

Algorithm 9: nonInteference()

```
for all pairs of rules (p1, e1) ∈ R and (p2, e2) ∈ R do
    if interferes((p1, e1), (p2, e2)) then
        return FALSE;
    end
end
return TRUE;
```

Corollary 3 (to Theorem 9)**.** *If all forced rules in the system contain only positive literals in their effects and state variables occur only positively in their preconditions, or if all forced rules contain only negative literals in their effects and state variables occur only negatively in their preconditions, then the system is terminating.*

Corollary 4 (to Theorem 11)**.** *If all forced rules in the system contain only positive literals in their effects and state variables occur only positively in their preconditions, or if all forced rules contain only negative literals in their effects and state variables occur only negatively in their preconditions, then the system is confluent.*

If the literals in the forced rules are only positive, they cannot negate any other literal in any other forced rule, nor can they falsify precondition of any other forced rule if the state variable in the preconditions also occur positively. Same is true for rules with only negative literals and state variable occurring negatively. As the system has only non-interfering rules, the system is confluent and it also terminates.

One important thing to note here is that lack of interference is a sufficient but not a necessary condition for confluence or termination. For example, consider a rule model with two forced rules $(a, \neg a)$ and $(b, \neg a)$. If the system starts in state where $v(a) = 1$ and $v(b) = 1$, the system is confluent and terminating, but the rules are interfering.

6.2 Rule Model with Interfering Rules

If the forced rules in the system are interfering with each other then determining if the system is confluent or terminating is more difficult. We need to employ some other techniques and methods.

The first thing we can do to reduce the difficulty of the problem is to eliminate rules which can never be fired from the rule model. But as proven below, this problem is

not tractable.

Theorem 13. *Testing if a forced rule in the rule model is never applicable is PSPACE-complete.*

Proof Idea. To prove that this problem is in PSPACE, we can reduce it to reachability (or planning) problem, which has been proven to be in PSPACE [8]. We can check if a state in which the given rule can be applicable is reachable from the initial state or from some other e-reachable state. To prove that this problem is PSPACE-hard, we need to reduce any polynomial space bounded Turing machine to our problem in polynomial time. We can do so by encoding transitions of the Turing machine to our rule model similar to how it was done in Section 4 for Lemma 2 and Lemma 6. We can then introduce a rule which will only be applicable if the Turing machine reaches the rejecting state. If the Turing machine reaches the accepting state the rule should never be applicable.

Proof. Algorithm 10 takes a rule (p, e) as input and checks if it is applicable in any state reachable from the initial state or from any e-reachable state. If the algorithm returns TRUE, the rule can never be applicable in any execution. We use the PSPACE Algorithm 4, SREACH (s_1, s_2, t) to test this. The call SREACH(s_1, s_2, t) takes into account the stable states and uses event rules to find successor of those states. For a system with n state variables, 2^n is the total number of states and the set of all possible states of the system is given by $\mathcal{S}$. Testing applicability of a rule takes a fixed amount of space as we only need to store the current state and the rule. As all parts of the algorithm use only polynomial amount of space, the algorithm works in PSPACE.

Algorithm 10: CheckRule$((p, e))$

```
for all states s in S do
    /* Check if s is reachable from I.                    */
    if SREACH (I, s, 2^n) then
        /* Check if the rule is applicable                */
        if s ⊨ p then
            /* Check if effect changes the state          */
            if succ_e(s) ≠ s then
                return FALSE:
            end
        end
    end
end
return TRUE;
```

For PSPACE-hardness proof, we need to reduce a polynomial space bounded Turing

machine to our problem in polynomial time.

We will use the same polynomial space bounded deterministic Turing machine M, used in Lemma 2 and Lemma 6, which decides languages in PSPACE. We will also use the same procedure as in those proofs to encode the transitions of the Turing machine to our rule model.

For M, let n be the length of the tape. So position of the head can be at any place $0, 1, \ldots, n-1$. If the length of input string to the Turing machine is k, n is a polynomial function $f(k)$. We define state variables as

- $in(i, x)$ = symbol x is in tape position i, $x \in \Gamma$, and
- $at(i, q)$ = the Turing machine is in state q and the head is in position i.

So if the Turing machine contains input on the tape from position 1 to k, the initial configuration of the Turing machine can be encoded in our rule model as,

$I = at(0, q_0),\ in(0, \#),\ in(1, x_1),\ in(2, x_2), \ldots,\ in(k, x_k),$
$in(k+1, \#),\ in(k+2, \#), \ldots, in(n-1, \#)$ variables are TRUE
. . . and all other state variable are FALSE.

The rules $(p_{i,q,x}, e_{i,q,x})$ are also defined similarly as in the previous proofs. We define only forced rules and keep the event rule set empty. For Turing machine in state q, the machine head at position $i < n-1$, and the head reading alphabet x at i^{th} position, the transition to replace x with y, move right and be in state q' can be encoded to our rule model in a forced rule (p, e) defined as follows:

$$p_{i,x,q} = in(i, x) \wedge at(i, q)$$
$$e_{i,x,y,q,q'} = \{\neg in(i, x),\ \neg at(i, q),\ in(i, y),\ at(i+1, q')\}$$

We can handle all transitions of the Turing machines using such rules as discussed in the proofs of Lemma 2 and Lemma 6. We limit the value of i to $i < n-1$ and $i > 0$, for rules corresponding to right and left movement of the head respectively, to handle boundary conditions.

Now, we will add an additional forced rule in our model from q_{reject}. The state q_{reject} is the rejecting state of the Turing machine where it halts while rejecting the input. We assume that the Turing machine moves its head all the way back to the 0^{th} position and clears the entire tape when it reaches the rejecting state before halting, i.e., $at(0, q_{reject})$ is TRUE and $in(i, \#)$ is TRUE for all possible values of i. To reduce this Turing machine to our problem of testing if a rule never fires, we will add one more transition rule from the rejecting state q_{reject} to a new state which is not reachable without firing this new rule. There are many options for this, but one of the easy ways to do this is to use a state in which all state variables are TRUE. This is because it guarantees that this state has never been reached before as any

other state reduced from the Turing machine cannot have a state variable $at(i, q_j)$ TRUE for more than one values of i or j.
So the new rule can then be :

$$p_{new} = at(0, q_{reject})$$
$$e_{new} = \bigcup_{i=0}^{n} \bigcup_{j=0}^{m} \{at(i, q_j)\} \cup \bigcup_{i=0}^{n} \bigcup_{x \in \Gamma} \{in(i, x)\}$$

This new rule will only be fired if the Turing machine reaches the rejecting state. So if we query this particular rule to our problem, it will correctly answer that the rule will never be fired if the Turing machine reaches the accepting state. The state variables in our rule model are defined using i, q and x. The values of i depend on the tape length n which is polynomial $f(k)$ in terms of input length k. Also the values which q can take depend on the number of states of the Turing machine which is polynomial in terms of the Turing machine description length $\|M\|$, and x belongs to the finite alphabet set Γ. As there are only polynomial number of i, q and x combinations, there can only be polynomial number of rules, which results in a polynomial time reduction of the Turing machine to our problem. So this reduction proves that the problem of testing if a rule in the rule model is never be applicable is PSPACE-hard.

As this problem is both in PSPACE and PSPACE-hard, it is PSPACE-complete. □

6.2.1 Approximation Method

As the complexity for finding which rules can be fired and which rules can never be fired with certainty is quite high, we can instead use an approximation method in PTIME for this. The approximation is given as follows:

1. Take any event rule, and let M be the set of literals in the effect of the event rule and literals entailed by the precondition which the effect does not change. An event can occur only in a stable state where none of the forced rules are applicable. If forced rules in the system are $(p_1, e_1), (p_2, e_2), ..., (p_n, e_n)$, then stable states satisfy the formula $(\neg p_1 \vee e_{l1}) \wedge (\neg p_2 \vee e_{l2}) \wedge ... \wedge (\neg p_n \vee e_{ln})$ where each $e_{li} = \bigwedge_{e_i} l$, for $1 \leq i \leq n$. Let all the literals logically entailed by this formula be a set N. Let $L_0 = M \cup N$.

   Literals entailed by a propositional formula can be found by a satisfiability algorithm, but the running time of any such algorithm is exponential in the worst case. Instead, such literals could be found by approximating satisfiability as follows: to see if l is a logical consequence of some set C of clauses, perform unit resolution with $C \cup \{\bar{l}\}$. If the empty clause is obtained, then l is a logical consequence of C. Unit resolution can be performed in linear time in the size of a clause set [22].

2. Define inductively a sequence of sets L_i for $i > 0$ as follows.
   $L_i = L_{i-1} \setminus \{\bar{l} \mid l \in e$ for some forced rule (p, e) such that $L_{i-1} \not\models \neg p\}$

3. If $L_i = L_{i+1}$ for some i, then the literals in L_i must hold forever, in other words, they cannot be made false by an execution sequence of any length. So, all rules (p, e) not falsified by L_i (i.e., $L_i \not\models \neg p$) may become applicable.

4. Repeat steps 1-3 for all event rules in the rule model. The forced rules which do not become applicable due to any of the events can be removed from further analysis.

This approximation works in polynomial time as there are only linear number of events, rules and literals in the size of the transition system description. It goes through each event and in the worst case needs to go through all rules for each literal. The approximation may or may not reduce the number of rules in the rule model.

Now we can perform analysis only on the forced rules which may become applicable, while safely ignoring other forced rules. Also as we have seen earlier, non-interfering rules do not affect the confluence of termination of the system. So we move our attention to interfering rules.

6.2.2 Clusters of Interfering Rules

Interfering rules might form clusters where rules may only interfere with some other rules in the same cluster, but not with rules outside that particular cluster. Finding and analysing these clusters individually can be easier than analysing the entire system at once. Note that we only discuss interference and non-interference with respect to forced rules.

Definition 31 (Cluster). A set c of forced rules is a *cluster* if no rule $(p, e) \in c$ interferes with any other rule $(p', e') \in \mathcal{R} \setminus c$ for the transition system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$.

We use the following Algorithm 11 to find the smallest possible clusters from the forced rule set $\mathcal{R}$ for the system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$ such that each forced rule belongs to at least one cluster. Each c_k is a cluster of interfering rules and the maximum value of k gives the maximum number of clusters that can be formed such that no rules in one cluster interfere with any rules in any other cluster. For a system in which all forced rules are non-interfering, if there are n forced rules then there would be maximum of n clusters that could be formed in such a way. This algorithm basically computes the equivalence classes induced by the interference relation, i.e., the reflexive transitive closure of the interference relation.

Algorithm 11: CLUSTER $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$

```
k := 1;
repeat
    Take a rule (p, e) ∈ R unassigned to any cluster c_l, l ≤ k;
    c_k := {(p, e)}; /* Assign the rule to a cluster                    */
    r := {}; /* set to keep track of checked rules                     */
    repeat
        Take any rule (p_i, e_i) ∈ c_k such that (p_i, e_i) ∉ r
        r := r ∪ {(p_i, e_i)};
        for all other rules (p_j, e_j) ∈ R and (p_j, e_j) ∉ any cluster c_l, l ≤ k do
            if interferes((p_i, e_i), (p_j, e_j)) then
                c_k := c_k ∪ {(p_j, e_j)};
            end
        end
    until c_k = r;
    k := k + 1;
until all rules in R are assigned a cluster;
return all clusters;
```

In the clusters obtained from this algorithm, all rules interfere with at least one other rule in the same cluster. This algorithm has to go through each rule to check if it interferes with any other rule. So if there are m rules, and n state variables, the time complexity of this algorithm is $\mathcal{O}(m^2n^2)$ as it also uses Algorithm 8 to test interference.

Rules in a cluster do not interfere with any rule outside their own cluster. That means union of two clusters is also a cluster as rules in the union would not interfere with any rules outside that the union.

Now non-interfering rules do not have any influence over termination as termination requires at least two rules with interfering effects. They also do not affect confluence as, from Lemma 8, if a non-interfering rule is fired in any maximal execution the literals in its effect hold in all maximal executions. So, as these rules have influence neither on termination nor on confluence, we must focus only on analysing the clusters with multiple interfering rules and analyse their properties.

6.3 Analysis with Clusters

In this subsection we will see how properties of clusters can be used to determine termination and confluence of the whole system under certain conditions. We first look at definitions of terminating clusters and confluent clusters.

Definition 32 (Terminating cluster). For a transition system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$, let $\mathcal{S}_e$ be the set of all e-reachable states. A cluster $c \subseteq \mathcal{R}$ is *terminating* if all executions starting from any e-reachable state $s_e \in \mathcal{S}_e$ using only rules $(p, e) \in c$ are terminating.

Definition 33 (Confluent cluster). For a transition system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$, let $\mathcal{S}_e$ be the set of all e-reachable states. A cluster $c \subseteq \mathcal{R}$ is *confluent* if all executions starting from any e-reachable state $s_e \in \mathcal{S}_e$ using only rules $(p, e) \in c$ are confluent.

Definition 34 (Mutually non-interfering clusters). If no rule from cluster c_1 interferes with any rule from the cluster c_2, then c_1 and c_2 are *mutually non-interfering.*

Algorithm 11 gives a set of clusters which are all mutually non-interfering. Now take an example of a system with initial state $I = \{a, \neg b, \neg c, \neg d\}$, no event rules, and two mutually non-interfering clusters from the system, $c_1 = \{(a, \{b\})\}$ and $c_2 = \{(b, \{c\}), (\neg c, \{d\})\}$. Each cluster taken separately is confluent, but their union is not. This is because the rule in c_1 can make a rule in c_2 applicable. Similarly, for a system with initial state $I = \{a, \neg b, \neg c, \neg d\}$, no event rules, and two mutually non-interfering clusters from the system, $c_1 = \{(a, \{b\})\}$ and $c_2 = \{(b, \{c\}), (c, \{\neg c\})\}$, each cluster taken separately is terminating, but their union is not.

Lemma 14. *The union of two mutually non-interfering confluent clusters is confluent if for all variables x in the effects of the rules of one cluster, x does not occur in the precondition of the rules in the other cluster.*

Proof. For the system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$, let $c_1 \subseteq \mathcal{R}$ and $c_2 \subseteq \mathcal{R}$ be two mutually non-interfering confluent clusters such that for all x in the effects of the rules of one cluster, x does not occur in the precondition of the rules in the other cluster. Let $\mathcal{S}_e$ be the set of all e-reachable states for the system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$. For executions starting from any $s_e \in \mathcal{S}_e$, let s_{f1} be a reachable state in any execution using only rules from confluent cluster c_1 as c_1 is confluent. So for all x such that $s_e(x) \neq s_{f1}(x)$, $x \in e$ or $\neg x \in e$ for some $(p, e) \in c_1$. The rules in c_1 and c_2 do not interfere with each other and also according to the theorem statement, the effects of rules in one cluster cannot make TRUE the precondition of the rules in other cluster. So even if the rules from cluster c_2 are fired, it cannot make rules from c_1 applicable.

For the same $s_e \in \mathcal{S}_e$, let s_{f2} be a reachable state in any execution using only rules from c_2. Following the same reasoning, for all x such that $s_e(x) \neq s_{f2}(x)$, $x \in e$ or $\neg x \in e$ for some $(p, e) \in c_2$, and no rule in c_2 can be made applicable due to rules in c_1.

So for any execution starting from s_e using rules from $c_1 \cup c_2$, there exists a state s_f such that $s_f(x) = s_{f1}(x)$ for all x such that $s_e(x) \neq s_{f1}(x)$, and $s_f(x) = s_{f2}(x)$ for all x such that $s_e(x) \neq s_{f2}(x)$. As a result, the system is confluent with s_f being

reachable from all reachable states for executions starting from s_e.

This result is valid for executions starting from any e-reachable state. Hence, the union of two mutually non-interfering confluent clusters is a confluent cluster if for all variables x in the effects of the rules of one cluster, x does not occur in the precondition of the rules in the other cluster. □

Theorem 15. *The union of any number of mutually non-interfering confluent clusters is confluent if for all x in the effects of the rules of any cluster, x does not occur in the precondition of the rules in any other cluster.*

Proof. The proof is by mathematical induction in the number of clusters i.

Induction hypothesis: For i mutually non-interfering confluent clusters such that for all x in the effects of the rules of any cluster, x does not occur in the precondition of the rules in any other cluster, their union is confluent.

Base case $i = 1$: For one confluent cluster, it is trivial as the rules set is already confluent.

Inductive step $i \geq 1$: Lemma 14 proves this for the union of two clusters. Using the induction hypothesis, let $c_1, c_2, \ldots, c_i$ be the i mutually non-interfering confluent clusters such that their union is also a confluent cluster. We need to prove that union of this cluster with $i + 1^{th}$ non-interfering confluent cluster c_{i+1} is confluent if for all x in the effects of the rules of any cluster, x does not occur in the precondition of the rules in any other cluster.

Let $c = c_1 \cup c_2 \cup \cdots \cup c_i$. This is confluent and as all clusters are mutually non-interfering, we can treat this as one cluster whose rules do not interfere with rules in c_{i+1}. Also as no variable x in the effects of the rules of one cluster occurs in the precondition of the rules in the other cluster for both c and c_{i+1}, we can say that $c \cup c_{i+1}$ is confluent from Lemma 14.

Hence the union of any number of mutually non-interfering confluent clusters is confluent if for all x in the effects of the rules of any cluster, x does not occur in the precondition of the rules in any other cluster. □

Lemma 16. *The union of two mutually non-interfering terminating clusters is terminating if for all variables x in the effects of the rules of one cluster, x does not occur in the precondition of any rule in any other cluster.*

Proof. For the system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$, let $c_1 \subseteq \mathcal{R}$ and $c_2 \subseteq \mathcal{R}$ be two mutually non-interfering terminating clusters such that for all x in the effects of the rules of one

cluster, x does not occur in the precondition of the rules in the other cluster. Let $\mathcal{S}_e$ be the set of all e-reachable states for the system $(\mathcal{A}, \mathcal{R}, \mathcal{U}, I)$. The rules in c_1 and c_2 do not interfere with each other and also according to the theorem statement, the effects of rules in one cluster cannot make TRUE the precondition of the rules in other cluster. This means rules in one cluster cannot make the rules in other cluster applicable. Also both clusters are terminating. So for any execution from e-reachable state $s_e \in \mathcal{S}_e$ only finite number of rules from c_1 will be fired, as rules from c_2 cannot make any additional rule from c_1 applicable. Similarly, for any execution from the same e-reachable state $s_e \in \mathcal{S}_e$ only finite number of rules from c_2 will be fired, as rules from c_1 cannot make any additional rule from c_2 applicable. So only finite number of rules will be fired in any execution starting from any e-reachable state using rules from both c_1 and c_2. This result is valid for executions starting from any e-reachable states. Hence, $c_1 \cup c_2$ is terminating. □

Theorem 17. *The union of any number of mutually non-interfering terminating clusters is terminating if for all variables x in the effects of the rules of one cluster, x does not occur in the precondition of any rule in the other cluster.*

Proof. The proof is by mathematical induction in the number of clusters i.

Induction hypothesis: For i mutually non-interfering terminating clusters such that for all x in the effects of the rules of any of the clusters, x does not occur in the precondition of the rules in any other cluster, their union is terminating.

Base case $i = 1$: For one terminating cluster, it is trivial as the rules set is already terminating.

Inductive step $i \geq 1$: Lemma 16 proves this for the union of two clusters. Using the induction hypothesis, let $c_1, c_2, \ldots, c_i$ be the i mutually non-interfering terminating clusters such that their union is also a terminating cluster. We need to prove that union of this cluster with $i + 1^{th}$ non-interfering terminating cluster c_{i+1} is also terminating if for all x in the effects of the rules of any cluster, x does not occur in the precondition of the rules in any other cluster.

Let $c = c_1 \cup c_2 \cup \cdots \cup c_i$. This is terminating and as all clusters are mutually non-interfering, we can treat this as one cluster whose rules do not interfere with rules in c_{i+1}. Also as no variable x in the effects of the rules of one cluster occur in the precondition of the rules in the other cluster for both c and c_{i+1}, we can say that $c \cup c_{i+1}$ is terminating from Lemma 16.

Hence the union of any number of mutually non-interfering confluent clusters is terminating if for all x in the effects of the rules of any cluster, x does not occur in the precondition of the rules in any other cluster. □

Testing if the clusters are internally confluent or terminating is still in PSPACE. The approximation method and finding clusters of interfering rules may bring the difficulty of the problem low enough that it can be tested in reasonable time, but the problem is still PSPACE-complete. In the worst case, the approximations reduce nothing and the entire rule set is just one big cluster, which is exactly our original problem. Also the clusters can be internally terminating, but not confluent or confluent but not terminating. These properties are not related with one another as mentioned before.

7 Conclusion and Future Work

This thesis has been a preliminary investigation for rule analytics, mainly confluence and termination problems, pertaining to condition-effect rules for the EIAI project. The focus in this thesis was on a rule model with only Boolean facts as datatype. Confluence and termination problems for such kind of rule model are PSPACE-complete. We also looked into certain practical aspects such as events which do not affect the PSPACE complexity of these problems. Finding bounds, however, is in PSPACE for a rule model with events, while it is in P and NP if we limit the analysis to constant and polynomial bounds respectively, for a rule model without events. Finally, we presented algorithms for some tractable cases which can be applied in practical applications. A rule model with all non-interfering rules can be analyzed in PTIME, while rule model with interfering rules is much more complex. We discussed an approximation method to find out which rules can be fired in the system and clustering method to make the problems easier to solve for certain rule models. This section is not exhaustive and more tractable cases should be studied in future.

The next step is to analyse these problems for a rule model with conditional effects consisting of if-else statements and then move to more complex rule models with other datatypes such as integers and reals.

Another important aspect to look into is incremental method of analysis. Currently all analysis has to be done again for any modification such as addition of a rule or removal of a rule in the rule model. In future, we would like to look into methods in which the previous analysis result still stays valid and supplements in speeding the analyses of the modified rule model. We would also like to look into convergent systems, which are both terminating and confluent, and study if checking one of the termination or confluent properties helps deciding the other, i.e., to test if there is any relationship between termination and confluence.

Priority among rules is something we have not discussed in this thesis. The rules in our rule model do not have any inherent priority. Our focus has been on non-determinism in the execution sequence. One of the applicable rules is non-deterministically chosen to fire. But if there is some ordered priority among applicable rules in each state of the system, the rules can only be fired in a certain order (or some rules may not fire at all). This brings some determinism to the execution and the system might be made both terminating and confluent based on how the priorities are assigned to the rules. This is something that can be studied further especially for intractable cases in the presence of interfering rules.

References

[1] A. Aiken, J. Widom, and J. M. Hellerstein, "Behavior of database production rules: termination, confluence, and observable determinism", *Proceeding of the ACM SIGMOD International Conference on Management of Data*, pp. 59-68, 1992.

[2] C. Baier, and J. P. Katoen, "Principles of model checking", MIT Press, 2008.

[3] J. Bailey, G. Dong, and K. Ramamohanarao, "Decidability and undecidability results for the termination problem of active database rules", *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 264-273, 1998.

[4] E. Best, and K. Voss, "Free choice systems have home states", *Acta Informatica* 21, pp. 89–100, 1984.

[5] E. Best, J. Esparza, "Existence of home states in Petri nets is decidable", *Information Processing Letters* 116(6), pp. 423-427, 2016.

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs", *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer-Verlag, pp. 193-207, 1999.

[7] A. L. Blum, and M. L. Furst, "Fast planning through planning graph analysis", *Artificial Intelligence* 90(1-2), pp. 281-300, 1997.

[8] T. Bylander, "The computational complexity of propositional STRIPS planning", *Artificial Intelligence* 69(1-2), pp. 165-204, 1994.

[9] H. Carstensen, "Decidability questions for fairness in Petri nets", *Proceedings of 4th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science* 247, Springer-Verlag, pp. 396-407, 1987.

[10] L. Chavarría-Báez, and X. Li, "Termination analysis of active rules—A Petri net based approach", *Systems, Man and Cybernetics*, IEEE, pp. 2205-2210, 2009.

[11] A. Church, and J. B. Rosser, "Some properties of conversion", *Transactions of the American Mathematical Society* 39(3), pp. 472-482, 1936.

[12] E. M. Clarke, and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic", *Workshop on Logic of Programs*, Springer-Verlag, pp. 52-71, 1981

[13] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Transactions on Programming Languages and Systems* 8(2), pp. 244-263, 1986.

[14] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving", *Formal Methods in System Design* 19(1), pp. 7-34, 2001.

[15] E. M. Clarke, and J. M. Wing, "Formal methods: state of the art and future directions", *ACM Computing Surveys* 28(4), pp. 626-643, 1996.

[16] S. Comai, and L. Tanca, "Termination and confluence by rule prioritization", *IEEE Transactions on Knowledge and Data Engineering* 15(2), pp. 257-270, 2003.

[17] H. Comon, G. Godoy, and R. Nieuwenhuis, "The confluence of ground term rewrite systems is decidable in polynomial time." *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science FOCS*, pp. 298-307, 2001.

[18] J. C. Corbett, "Evaluating deadlock detection methods for concurrent software", *IEEE Transactions on Software Engineering* 22(3), pp. 161-180, 1996.

[19] M. Dauchet, and S. Tison, "The theory of ground rewrite systems is decidable", *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pp. 242-248, 1990.

[20] N. Dershowitz, "A taste of rewrite systems", *Functional Programming, Concurrency, Simulation and Automated Reasoning. Lecture Notes in Computer Science* 693, Springer-Verlag, pp. 199-228, 1993.

[21] N. Dershowitz, and D. A. Plaisted, "Rewriting", *Chapter 9 in: Handbook of Automated Reasoning* 1, A. Robinson, A. Voronkov (Eds.), Elsevier, pp. 535-610, 2001.

[22] W. F. Dowling, and J. H. Gallier, "Linear-time algorithms for testing the satisfiability of propositional Horn formulae", *The Journal of Logic Programming* 1(3), Elsevier, pp. 267-284, 1984.

[23] D. de Frutos Escrig, and C. Johnen, "Decidability of home space property", *Université de Paris-Sud, Centre d'Orsay, Laboratoire de Recherche en Informatique*, 1989.

[24] J. Ezpeleta, J. M. Colom, and J. Martinez, "A Petri net based deadlock prevention policy for flexible manufacturing systems", *IEEE Transactions on Robotics and Automation* 11(2), pp. 173-184, 1995.

[25] R. E. Fikes, and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving", *Artificial Intelligence* 2(3-4), pp. 189-208, 1971.

[26] R. R. Howell, L. E. Rosier, and H. C. Yen, "A taxonomy of fairness and temporal logic problems for Petri nets", *Theoretical Computer Science* 82(2), pp. 341-372, 1991.

[27] G. Huet, and D. Lankford, "On the uniform halting problem for term rewriting systems", *INRIA Technical Report* 283, 1978.

[28] H. V. Jagadish, A. O. Mendelzon, and I. S. Mumick, "Managing conflicts between rules", *Journal of Computer and System Sciences* 58(1), pp. 13-28, 1999.

[29] X. Jin, Y. Lembachar, and G. Ciardo, "Symbolic termination and confluence checking for ECA rules", *Transactions on Petri Nets and Other Models of Concurrency IX*, Springer-Verlag, pp. 99-123, 2014.

[30] S. Kaplan, "Simplifying conditional term rewriting systems: unification, termination and confluence." *Journal of Symbolic Computation* 4(3), pp. 295-334, 1987.

[31] H. Kautz, and B. Selman, "Pushing the envelope: planning, propositional logic, and stochastic search", *Proceedings of the Thirteenth National Conference on Artificial Intelligence* 2, AAAI Press, pp. 1194-1201, 1996.

[32] I. Leahu, and F. Tiplea, "The confluence property for Petri nets and its applications", *Proceedings of Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing SYNASC*, IEEE, pp. 430-436, 2006.

[33] Z. Manna, and A. Pnueli, "Temporal verification of reactive systems: safety", Springer-Verlag, 1995.

[34] T. Murata, "Petri nets: properties, analysis and applications", *Proceedings of the IEEE* 77(4), pp. 541-580, 1989.

[35] M. Oyamaguchi, "The Church-Rosser property for ground term-rewriting systems is decidable", *Theoretical Computer Science* 49(1), pp. 43-79, 1987.

[36] C. H. Papadimitriou, "Computational Complexity", Addison-Wesley Publishing Company, 1994.

[37] C. A. Petri, "Kommunikation mit Automaten", *PhD thesis*, University of Bonn, West Germany, 1962.

[38] D. Plump, "Termination of graph rewriting is undecidable", *Fundamenta Informaticae* 33(2), pp. 201-209, 1998.

[39] J. Rintanen, "Introduction to automated planning", *Lecture notes of the AI planning course*, Albert-Ludwigs-University Freiburg, 2006.

[40] J. Rintanen, "Diagnosers and diagnosability of succinct transition systems.", *Proceedings of 20th International Joint Conference on Artificial Intelligence*, AAAI Press, pp. 538-544, 2007.

[41] W. J. Savitch, "Relationship between nondeterministic and deterministic tape complexities", *Journal of Computer and System Sciences* 4(2), pp. 117-192, 1970.

[42] M. Sipser, "Introduction to the Theory of Computation", Cengage Learning, 2013.

[43] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis", *ACM SIGPLAN Notices* 45(1), pp. 313-326, 2010.

[44] L. van der Voort, and A. Siebes, "Termination and confluence of rule execution", *Proceedings of the Second International Conference on Information and Knowledge Management*, ACM, pp. 245-255, 1993.

[45] X. Wang, J.-H. You, and L. Y. Yuan, "On confluence property of active databases with meta-rules", *International Workshop on Rules in Database Systems, Lecture Notes in Computer Science* 1312, Springer-Verlag, pp. 118–132, 1997.

[46] D. Zimmer, A. Meckenstock, and R. Unland, "Using Petri nets for rule termination analysis", *Proceedings of the Workshop on Databases: Active and Real-Time*, ACM, pp. 29-32, 1996.

List of definitions